IBM WebSphere Application Server for z/OS, Version 8.5

Developing WebSphere applications

SA32-1083-00

Note

Before using this information, be sure to read the general information under “Notices” on page 2243.

Compilation date: June 1, 2012

Contents

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an email form appears.
  3. Fill out the email form as instructed, and submit your feedback.
- To send comments on PDF books, you can email your comments to: **wasdoc@us.ibm.com**.

  Your comment should pertain to specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer. When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about your comments.

Using this PDF

Links

Because the content within this PDF is designed for an online information center deliverable, you might experience broken links. You can expect the following link behavior within this PDF:

- Links to Web addresses beginning with http:// work.
- Links that refer to specific page numbers within the same PDF book work.
- The remaining links will *not* work. You receive an error message when you click them.

Print sections directly from the information center navigation

PDF books are provided as a convenience format for easy printing, reading, and offline use. The information center is the official delivery format for IBM WebSphere Application Server documentation. If you use the PDF books primarily for convenient printing, it is now easier to print various parts of the information center as needed, quickly and directly from the information center navigation tree.

To print a section of the information center navigation:

1. Hover your cursor over an entry in the information center navigation until the **Open Quick Menu** icon is displayed beside the entry.
2. Right-click the icon to display a menu for printing or searching your selected section of the navigation tree.
3. If you select **Print this topic and subtopics** from the menu, the selected section is launched in a separate browser window as one HTML file. The HTML file includes each of the topics in the section, with a table of contents at the top.
4. Print the HTML file.

For performance reasons, the number of topics you can print at one time is limited. You are notified if your selection contains too many topics. If the current limit is too restrictive, use the feedback link to suggest a preferable limit. The feedback link is available at the end of most information center pages.

Chapter 1. Developing ActivitySessions

This page provides a starting point for finding information about ActivitySessions, a WebSphere extension for reducing the complexity of commitment rules and limitations that are associated with one-phase commit resources.

Use ActivitySessions to extend the scope and group multiple local transactions. With this capability, you can commit these transactions based on either deployment criteria or through explicit program logic.

Developing an enterprise application to use ActivitySessions

This topic provides an overview of the high-level tasks for using ActivitySessions in enterprise applications.

About this task

Before you use ActivitySessions in enterprise applications, consider the following points:

- An application that is accessed under an ActivitySession context can receive a javax.transaction.InvalidTransactionException RemoteException, thrown by the Enterprise JavaBeans (EJB) container when servicing any application method. This exception occurs when an instance of an enterprise bean that has an ActivitySession-based activation policy becomes involved with concurrent global and local transactions.
- To enable an enterprise bean to participate in an ActivitySession context and support ActivitySession-based operations, it must be configured with an ActivationPolicy of ACTIVITY_SESSION. A bean configured with ActivationPolicy of either TRANSACTION or ONCE cannot participate in an ActivitySession context.
- A session bean can either use container-managed ActivitySessions or implement bean-managed ActivitySessions; entity beans can use only container-managed ActivitySessions. A bean is deployed to be bean-managed or container-managed with respect to ActivitySession management by setting its transaction type deployment attribute to be bean-managed or container-managed when you deploy the enterprise bean. A bean that uses bean-managed transactions can use bean-managed ActivitySessions; a bean that uses container-managed transactions can use container-managed ActivitySessions.
- If you want a session bean or an enterprise application client to manage its own ActivitySessions, you must write the code that explicitly demarcates the boundaries of an ActivitySession, as described in Developing an enterprise bean or J2EE client to manage ActivitySessions.

The following high level tasks illustrate how to use an ActivitySession in an enterprise application:

Procedure

- Develop an enterprise application that uses one or more enterprise beans that are persisted to non-transactional data stores. Use this approach for an application that needs to coordinate multiple one-phase resource managers, for example, for two or more entity enterprise beans whose persistence is delegated to LocalTransaction resource adapters.

  In this scenario, the enterprise beans that the application uses have an Activation policy of ActivitySession and a local transaction containment policy with a boundary of ActivitySession and resolution-control of ContainerAtBoundary. The container synchronizes the EJB state data with the one-phase resource managers at ActivitySession completion, and no application code needs to be aware of ActivitySession support.
- Develop an enterprise application in which an enterprise bean accesses a resource manager multiple times in different business methods. Use this approach for an application that needs to extend a resource manager local transaction (RMLT) over several business methods of an enterprise bean instance.

  In this scenario, the enterprise beans that the application uses have an Activation policy of ActivitySession and a local transaction containment policy with a boundary of ActivitySession and resolution-control of Application. The application logic starts and ends the RMLTs, for example, using the

javax.resource.cci.LocalTransaction interface offered by a LocalTransaction Connector, but is not constrained to start and commit the LocalTransaction in the same method.

- Develop an enterprise client application to use an ActivitySession to scope EJB activation and load-balancing. Use this approach for an application client that needs to access an entity bean instance several times in the same client session, either without needing to run under a transaction context, or with the need to run under a number of distinct and serially-executed transactions.

  In this scenario, the enterprise beans that the application uses have an Activation policy of ActivitySession and a local transaction containment policy appropriate to the function of the enterprise bean. The enterprise client application can represent a period of user activity, for example a signon period, during which a number of interactions occur with one or more enterprise beans. If the enterprise client application begins an ActivitySession and invokes the enterprise beans within the scope of the unit of work (UOW) that the ActivitySession represents, the container on the ActivitySession boundary activates the enterprise bean instances. The instances remain in the active state until the container passivates them at the end of the ActivitySession. Workload affinity management based on the ActivitySession is a platform quality of service. Global transactions can begin and end within the ActivitySession, if they are wholly encapsulated by the ActivitySession and run serially. EJB instances that are activated at the ActivitySession boundary remain active across the serial global transactions.
- Develop a Web application client to participate in an ActivitySession context. A Web application that runs in the WebSphere® Web container can participate in an ActivitySession context. Web applications can use the UserActivitySession interface to begin and end an ActivitySession context. Also, the ActivitySession can be associated with an HttpSession, thereby extending access to the ActivitySession over multiple HTTP invocations and supporting EJB activation periods that can be determined by the lifecycle of the Web HTTP client.

  The Web container manages ActivitySessions based on deployment descriptor attributes associated with the Web application module.

Example

For examples of using ActivitySessions in enterprise applications, see the topic about ActivitySessions samples.

Developing an enterprise bean or enterprise application client to manage ActivitySessions

Use this task to write the code needed by a session EJB or enterprise application client to manage an ActivitySession, based on the example code extract provided.

About this task

In most situations, an enterprise bean can depend on the EJB container to manage ActivitySessions within the bean. In these situations, all you need to do is set the appropriate ActivitySession attributes in the EJB module deployment descriptor, as described in the topic about configuring EJB module ActivitySession deployment attributes.. Further, in general, it is practical to design your enterprise beans so that all ActivitySession management is handled at the enterprise bean level.

However, in some cases you may need to have a session bean or enterprise application client participate directly in ActivitySessions. You then need to write the code needed by the session bean or enterprise application client to manage its own ActivitySessions.

Note: Session beans that use BMT and have an **Activate at** setting of `Activity session` can manage ActivitySessions. Entity beans cannot manage ActivitySessions; the EJB container always manages ActivitySessions within entity beans.

When preparing to write code needed by a session bean or enterprise application client to manage ActivitySessions, consider the points described in the topic about ActivitySession and transaction contexts.

To write the code needed by a session EJB or enterprise application client to manage an ActivitySession, complete the following steps, based on the following example code extract.

Procedure

1. Get an initial context for the ActivitySession.
2. Get an implementation of the UserActivitySession interface, by a JNDI lookup of the URL java:comp/websphere/UserActivitySession. The UserActivitySession interface is used to begin and end ActivitySessions and to query various attributes of the active ActivitySession associated with the thread.
3. Set the timeout, in seconds, after which any subsequently started ActivitySessions are automatically completed by the ActivitySession service. If the session bean or enterprise application client does not specifically set this value, the default timeout (300 seconds) is used.

   The default timeout can also be overridden for each application server, on the ***server*-> Activity Session Service** panel of the administrative console.
4. Start the ActivitySession, by calling the beginSession() method of the UserActivitySession.
5. Within the ActivitySession, call business methods to do the work needed. You can also call other methods of UserActivitySession to manage the ActivitySession; for example, to get the status of the ActivitySession or to checkpoint all the ActivitySession resources involved in the ActivitySession.
6. End the ActivitySession, by calling the endSession() method of the UserActivitySession.

Example

The following code extract provides a basic example of using the UserActivitySession interface:

```
// Get initial context
  InitialContext ic = new InitialContext();
// Lookup UserActivitySession
  UserActivitySession uas =
 (UserActivitySession)ic.lookup("java:comp/websphere/UserActivitySession");

// Set the ActivitySession timeout to 60 seconds
  uas.setSessionTimeout(60);
// Start a new ActivitySession context
  uas.beginSession();
// Do some work under this context
  MyBeanA beanA.doSomething();
  ...
  MyBeanB beanB.doSomethingElse();
// End the context
  uas.endSession(EndModeCheckpoint);
```

ActivitySession service application programming interfaces

The ActivitySession service provides an application programming interface that is available to Web applications, session Enterprise JavaBeans (EJBs), and Java platform for enterprise applications client applications for application-managed demarcation of ActivitySession context.

Applications use the UserActivitySession interface, which provides demarcation scope methods.

ActivitySession API

The ActivitySession service provides the UserActivitySession interface for use by EJB Session beans using bean-managed context demarcation, Web application components that are configured with the **ActivitySession control** attribute set to `Web Application`, and Java platform for enterprise applications client applications. This UserActivitySession interface defines the set of ActivitySession operations that are available to an application component. To obtain an implementation of this interface, use a Java Naming and Directory Interface (JNDI) lookup of the URL `java:comp/websphere/UserActivitySession`. The UserActivitySession interface is used to begin and end ActivitySessions and to query various attributes of the active ActivitySession that is associated with the thread.

For more information about the ActivitySession API, see the application programming interface (API) reference information.

The ActivitySession API and the implementation of its interfaces is contained in the com.ibm.websphere.ActivitySession package.

Programming Examples

The following code extract provides a basic example of using the UserActivitySession interface:

```
// Get initial context
  InitialContext ic = new InitialContext();
// Lookup UserActivitySession
  UserActivitySession uas =
  (UserActivitySession)ic.lookup("java:comp/websphere/UserActivitySession");

// Set the ActivitySession timeout to 60 seconds
  uas.setSessionTimeout(60);
// Start a new ActivitySession context
  uas.beginSession();
// Do some work under this context
  MyBeanA beanA.doSomething();
  ...
  MyBeanB beanB.doSomethingElse();
// End the context
  uas.endSession(EndModeCheckpoint);
```

Assembling applications that use ActivitySessions

You can set the ActivitySession deployment attributes for an enterprise bean or a Web application.

About this task

For an enterprise bean, you can set the ActivitySession deployment attributes so that the bean can participate in an ActivitySession context and support ActivitySession-based operations.

For a Web application, you can set the ActivitySession deployment attributes so that the application can start UserActivitySessions and perform work scoped within ActivitySessions.

Procedure

- Set EJB module ActivitySession deployment attributes.
- Set Web module ActivitySession deployment attributes.

Setting EJB module ActivitySession deployment attributes

Use this task to set the ActivitySession deployment attributes for an enterprise bean to enable the bean to participate in an ActivitySession context and support ActivitySession-based operations.

Before you begin

This task description assumes that you have an Enterprise Archive (EAR) file, which contains an application enterprise bean that can be deployed in WebSphere Application Server. For more details, see the topic about assembling applications.

About this task

You configure the deployment attributes of an application by using an assembly tool. This topic describes the use of Rational® Application Developer to configure the ActivitySession deployment attributes. These attributes are in addition to other deployment attributes, for example, “Load at”, which specifies when the bean loads its state from the database. For details about the fields in the assembly tool, and for associated task help, refer to the Rational Application Developer information.

To set the ActivitySession deployment attributes for an enterprise bean, complete the following steps:

Procedure

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the application EAR file.

   Note: Ensure that you set the target server as WebSphere Application Server Version 7.0.

   For example, to change attributes of an existing application, use the Import wizard to import the EAR file into the assembly tool. To start the Import wizard:

   a. Click **File** > **Import** > **EAR file**.

   b. Click **Next**, then select the EAR file.

   c. In the Target server field, select `WebSphere Application Server v7.0`.

   d. Click **Finish**.
3. In the Project Explorer view of the Java EE perspective, right-click the EJB module for the enterprise bean instance, then click **Open With** > **Deployment Descriptor Editor**. A property dialog notebook for the enterprise bean instance is displayed in the property pane.
4. In the property pane, select the Beans tab.
5. Select the bean that you want to change.
6. In the WebSphere Extensions section, under **Bean Cache**, set the **Activate at** attribute to **ActivitySession**:

   An enterprise bean with this activation policy is activated and passivated as follows:
   - On an ActivitySession boundary, if an ActivitySession context is present on activation.
   - On a transaction boundary, if a transaction context, but no ActivitySession context, is present on activation.
   - Otherwise, on an invocation boundary.
7. In the Local Transactions group box, set the **Boundary** attribute to **ActivitySession**: When this setting is used, the local transaction must be resolved within the scope of any ActivitySession in which it was started or, if no ActivitySession context is present, within the same bean method in which it was started.

   A setting of ActivitySession does not apply to any EJB home methods, for example, create or finder methods. EJB home methods cannot participate in an ActivitySession because this situation might cause deadlocks.
8. For entity beans, or session beans, set the ActivitySession properties for each EJB method.

   a. In the property pane, select the ActivitySession tab.

   b. In the **Configure ActivitySession policies** field, click **Add** or **Edit** to set the **ActivitySession kind** attribute for methods of the enterprise bean. This specifies how the container must manage the ActivitySession boundaries when delegating a method invocation to an enterprise bean's business method:

      Never The container invokes bean methods without an ActivitySession context.
      - If the client invokes a bean method from within an ActivitySession context, the container throws an InvalidActivityException exception, which is a javax.rmi.RemoteException.
      - If the client invokes a bean method from outside an ActivitySession context, the container behaves in the same way as if the **Not Supported** value was set. The client must call the method without an ActivitySession context.

      Mandatory
      The container always invokes the bean method within the ActivitySession context associated with the client. If the client attempts to invoke the bean method without an ActivitySession context, the container throws an ActivityRequiredException exception to the client. The ActivitySession context is passed to any EJB object or resource accessed by an enterprise bean method.

The ActivityRequiredException exception is javax.rmi.RemoteException.

Requires new

The container always invokes the bean method within a new ActivitySession context, regardless of whether the client invokes the method within or outside an ActivitySession context. The new ActivitySession context is passed to any enterprise bean objects or resources that are used by this bean method.

Any received ActivitySession context is suspended for the duration of the method and resumed after the method ends. The container starts a new ActivitySession before method dispatch and completes it after the method ends.

Required

The container invokes the bean method within an ActivitySession context. If a client invokes a bean method from within an ActivitySession context, the container invokes the bean method within the client ActivitySession context. If a client invokes a bean method outside an ActivitySession context, the container creates a new ActivitySession context and invokes the bean method from within that context. The ActivitySession context is passed to any enterprise bean objects or resources that are used by this bean method.

Not supported

The container invokes bean methods without an ActivitySession context. If a client invokes a bean method from within an ActivitySession context, the container suspends the association between the ActivitySession and the current thread before invoking the method on the enterprise bean instance. The container then resumes the suspended association when the method invocation returns. The suspended ActivitySession context is not passed to any enterprise bean objects or resources that are used by this bean method.

Supports

If the client invokes the bean method within an ActivitySession, the container invokes the bean method within an ActivitySession context. If the client invokes the bean method without a ActivitySession context, the container invokes the bean method without an ActivitySession context. The ActivitySession context is passed to any enterprise bean objects or resources that are used by this bean method.

c. Click **Next**.

d. Select the methods to which the ActivitySession kind policy is to be applied.

e. Click **Finish**.

How the container manages the ActivitySession boundaries when delegating a method invocation depends on both the **ActivitySession kind** set here, and the **Container transaction type**, as described in the topic about configuring transactional deployment attributes. For more detail about the relationship between these two properties, see the topic about ActivitySession and transaction container policies in combination.

9. Save your changes to the deployment descriptor.
   a. Close the Deployment Descriptor Editor.
   b. When prompted, click **Yes** to save changes to the deployment descriptor.
10. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
11. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
12. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

    Important: Use **Run On Server** for unit testing only. The assembly tool controls the WebSphere Application Server installation and, when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application. For example, to use the administrative console, see the topic about deploying and administering enterprise applications.

Setting Web module ActivitySession deployment attributes

Use this task to set the ActivitySession deployment attributes for a Web application to start UserActivitySessions and perform work scoped within ActivitySessions.

Before you begin

This task assumes that you have an Enterprise Archive (EAR) file that contains an application enterprise bean that can be deployed in WebSphere Application Server. For more details, see the topic about assembling applications.

About this task

You can configure the deployment attributes of an application by using an assembly tool. This topic describes the use of Rational Application Developer to configure the deployment attributes.

To set the ActivitySession deployment attributes for a Web application, complete the following steps:

Procedure

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the Web module. For example, to change attributes of an existing module, click **File** > **Open**, then select the archive file for the module. For example, to change attributes of an existing module, use the Import wizard to import the EAR or WAR file into the assembly tool. To start the Import wizard:
   a. Click **File** > **Import**.
   b. Expand the Web folder, click **WAR file**, then click **Next**.
   c. Select the WAR file, then click **Finish**.
3. In the Project Explorer view of the Java EE perspective, right-click the component instance, right-click **Deployment Descriptor Editor**, then click **Open With** . A property dialog notebook for the Web module is displayed in the property pane.
4. In the property pane, select the Extended services tab.
5. Select the servlet that you want to change.
6. In the ActivitySession section, set the **ActivitySession control kind** attribute to Application, Container, or None.
   Application
   The Web application is responsible for starting and ending ActivitySessions, as follows:
   - If an HttpSession is active when an application begins an ActivitySession, the container associates the ActivitySession with the HttpSession.
   - If an ActivitySession is started in the absence of an HttpSession, the application must ensure it is completed before the dispatched method completes; otherwise, an exception results.
   - If an HttpSession is associated with a request dispatched to an application with this ActivitySession control value, and if that HttpSession has an ActivitySession associated with it, the container dispatches the request in the context of that ActivitySession. For example, the container resumes the ActivitySession context onto the thread before the dispatch.

- A Web application can use both transactions and ActivitySessions. Any transactions started within the scope of an ActivitySession must be ended by the Web component that started them and within the same request dispatch.

Container
A servlet has no access to UserActivitySessions. Any HttpSession started by the servlet has an ActivitySession automatically associated with it by the container, and this ActivitySession is put onto the thread of execution. If such a servlet is dispatched by a request that has an HttpSession containing no ActivitySession, then the container starts an ActivitySession and associates it with the HttpSession and the thread.

A Web application can use both transactions and ActivitySessions. Any transactions started within the scope of an ActivitySession must be ended by the Web component that started them and within the same request dispatch.

None A servlet has no access to UserActivitySession. An HttpSession started by the servlet does not have an ActivitySession automatically associated with it by the container. If such a servlet is dispatched by a request that has an HttpSession containing an ActivitySession, then the container dispatches the request in the context of that ActivitySession. For example, the container resumes the ActivitySession context onto the thread before the dispatch.

7. To apply the changes and close the assembly tool, click **OK**. Otherwise, to apply the values but keep the property dialog open for additional edits, click **Apply**.
8. Save your changes to the deployment descriptor.
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to save changes to the deployment descriptor.
9. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
10. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
11. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

Important: Use **Run On Server** for unit testing only. The assembly tool controls the WebSphere Application Server installation and, when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the WAR file. For example, to use the administrative console, see the topic about deploying and administering enterprise applications.

Chapter 2. Developing Application profiling

This page provides a starting point for finding information about application profiling, a WebSphere extension for defining strategies to dynamically control concurrency, prefetch, and read-ahead.

Application profiling and access intent provide a flexible method to fine-tune application performance for enterprise beans without impacting source code. Different enterprise beans, and even different methods in one enterprise bean, can have their own intent to access resources. Profiling the components based on their access intent increases performance in the application server run time.

Using the TaskNameManager interface

Using the TaskNameManager interface, you can programmatically set the current task name. It enables both overriding of the current task associated with the thread of execution and resetting of the current task with the original task.

About this task

Except for J2EE 1.3 applications that are running on a server where the 5.x Compatibility Mode attribute is selected, this interface cannot be used within Enterprise JavaBeans that are configured for container-managed transactions or container-managed ActivitySessions because units of work can only be associated with a task at the exact time that the unit of work is initiated. The call to set the task name must therefore be invoked before the unit of work is begun. Units of work cannot be named after they are begun. Calls on this interface during the execution of a container-managed unit of work are simply ignored.

Application profiling does not support queries of the task that is in operation at run time. Instead, applications interact with logical task names that are declaratively configured as application managed tasks. Logical references enable the actual task name to be changed without having to recompile applications.

Wherever possible, avoid setting tasks programmatically. The declarative method results in more portable function that can be easily adjusted without requiring redevelopment and recompilation.

Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

For a Version 6.0 client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to *true* in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the launchClient command.

Procedure

1. Configure application-managed tasks. Application profiling requires that a task name reference be declared for any task that is to be set programmatically. Task name references introduce a level of indirection so that the actual task set at run time can be adjusted by reassembly without requiring recoding or recompilation. Any attempt to set a task name that is undeclared as a task reference results in the raising of an exception. If a unit of work has already begun when a task name is set, then that existing unit of work is not associated with the task name. Only units of work that are begun after the task name is set are associated with the task.

Configure application-managed tasks as described in the following topics. To complete these tasks see the assembly tool information center:

- Configure application managed tasks for web components.
- Configure application managed tasks for application clients.
- Configure application managed tasks for Enterprise JavaBeans.

2. Perform a Java Naming and Directory Interface (JNDI) lookup on the TaskNameManager interface:

```
InitialContext ic = new InitialContext();
TaskNameManager tnManager = ic.lookup
("java:comp/websphere/AppProfile/TaskNameManager");
```

The *TaskNameManager* interface is not bound into the namespace if the application profiling service is disabled.

3. Set the task name:

```
try {
tnManager.setTaskName("updateAccount");
}
catch (IllegalTaskNameException e) {
// task name reference not configured. Handle error.
}
// . . .
//
```

The name passed to the setTaskName() method ("updateAccount" in this example) is actually a task name reference that you configured in step one.

4. Begin a UserTransaction

Note: If you are using a J2EE 1.3 application with the 5.x Compatibility mode set, the task name set in step 3 is now the active task name and you can disregard this step.

If you are using a J2EE application and the compatibility mode is not set, or if you are using a J2EE 1.4 application, you must begin a transaction for the task name to become active. A task name can only be associated with a transaction. Furthermore, it is associated with a transaction when that transaction is begun, and that task name is associated with the transaction for the life of the transaction. Therefore, the task name set above is not active at this point. You must begin a UserTransaction as the following code snippet illustrates:

```
try{
    InitialContext initCtx = new InitialContext();
    userTran = (UserTransaction) initCtx.lookup("java:comp/UserTransaction");
    userTran.begin();
}
catch(Exception e){
}
// . . .
//
```

Notice the resetTaskName() method on the TaskNameManager interface. Resetting the task name has no effect unless called by a J2EE 1.3 application running on a server for which the 5.x Compatibility Mode attribute is selected on the Application Profile Service's console page. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. A call to resetTask( ) should only be used by J2EE 1.3 applications when the 5.x Compatibility mode is set to undo the effects of any setTaskName() method operations and reestablish whatever task name was current when the component began execution. If the setTaskName() method has not been called, the resetTaskName() method has no effect.

TaskNameManager interface

The TaskNameManager is the programmatic interface to the application profiling function. Because on rare occasions it may be necessary to programmatically set the current task name, the TaskNameManager interface enables both overriding of the current task associated with the thread of execution and resetting of the current task with the original task.

Application profiling enables you to identify particular units of work to the WebSphere Application Server runtime environment. The run time can tailor its support to the exact requirements of that unit of work. Access intent is currently the only runtime component that makes use of the application profiling functionality. For example, you can configure one transaction to load an entity bean with strong update locks and configure another transaction to load the same entity bean without locks.

Application profiling introduces two concepts in order to achieve this function: tasks and profiles.

A *task* is a configurable name for a unit of work. *Unit of work* in this case means either a transaction or an ActivitySession.

A *profile* is simply a mapping of a task to a set of access intent policies that are configured on entity beans. When an invocation on a bean (whether by a finder method, a container managed relationship (CMR) getter, or a dynamic query) requires data to be retrieved from the back end system, the task of the active unit of work associated with the request is used to determine the exact requirement of the transaction. The same bean loads and behaves differently in the context of the task-to-profile mapping. Each profile provides the developer an opportunity to reconfigure the application's access intent.

Except for J2EE 1.3 applications that are executing on a server where the *5.x Compatibility Mode* attribute is selected, this interface cannot be used within Enterprise JavaBeans that are configured for container-managed transactions or container-managed ActivitySessions because units of work can only be associated with a task at the exact time that the unit of work is initiated. The call to set the task name must therefore be started before the unit of work is begun. Units of work cannot be named after they are begun. Calls on this interface during the execution of a container-managed unit of work are simply ignored.

The TaskNameManager interface is available to all J2EE components using the following Java Naming and Directory Interface (JNDI) lookup:

```
java:comp/websphere/AppProfile/TaskNameManager
```

```
package com.ibm.websphere.appprofile;

/**
* The TaskNameManager is the programmatic interface
* to the application profiling function. Using this interface,
* programmers can set the current task name on the
* thread of execution. The task name must have been
* configured in the deployment descriptors as a task
* reference associated with a task. The set task
* name's scope is the duration of the method
* invocation in the EJB and Web components and for
* the duration of the client process, or until the
* resetTaskName() method is invoked.
*/
public interface TaskNameManager {

/**
* Set the thread's current task name to the specified
* parameter. The task name must have been configured as
* a task reference with a corresponding task or the
* IllegalTaskName exception is thrown.
*/
public void setTaskName(String taskName) throws IllegalTaskNameException;

/**
```

```
* Sets the thread's task name to the value that was set
* at, or imported into, the beginning of the method
* invocation (for EJB and Web components) or process
* (for J2EE clients).
*/
public void resetTaskName();

}
```

Assembling applications for application profiling

To enable application profiling, you must configure tasks, create an application profile, and declaratively configure a unit of work on necessary methods.

Before you begin

Application profiling enables multiple access intent policies to be configured on the same entity bean, each specified for a particular unit of work. You can use the one of the default policies or create your own. To create your own access intent policy, see the topic, Creating a custom access intent policy, in the assembly tool information center.

Procedure

1. Configure tasks. Declaratively configure tasks as described in the following topics that are located in the assembly tool information center:
   - Configuring container-managed tasks for Enterprise Java Beans.
   - Configuring container-managed tasks for web components.
   - Configuring container-managed tasks for application clients.

   On rare occasions, you might find it necessary to configure tasks *programmatically*. Application profiling supports this requirement with a simple interface that enables a task name to be set before a unit of work is programmatically initiated. Setting a task name and then initiating a transaction or ActivitySession causes the task to be associated with the new unit of work. This interface cannot be used within Enterprise JavaBeans that are configured for container-managed transactions or container-managed ActivitySessions because units of work can only be associated with a task at the exact time that the unit of work is initiated. The call to set the task name must therefore be invoked before the unit of work is begun. Units of work cannot be named after they are begun. See the topic, Using the TaskNameManager interface.

   Note: If you select the 5.x Compatibility Mode attribute on the Application Profile Service's console page, then tasks configured on J2EE 1.3 applications are not necessarily associated with units of work and can arbitrarily be applied and overridden. This is not a recommended mode of operation and can lead to unexpected deadlocks during database access. Tasks are not communicated on requests between applications that are running under the Application Profiling 5.x Compatibility Mode and applications that are not running under the compatibility mode.

   For a Version 6.0 client to interact with applications run under the Application Profiling 5.x Compatibility Mode, you must set the *appprofileCompatibility* system property to *true* in the client process. You can do this by specifying the *-CCDappprofileCompatibility=true* option when invoking the launchClient command.
2. Create an application profile. See the assembly tool information center to complete this task.
3. Declaratively configure a unit of work on necessary methods. In step one of this article, you defined a task on a method. The task defined on a method only becomes active when a unit of work is begun on that method's behalf. The method must begin a new unit of work for the configured task to be applied. If the method runs under an imported unit of work, then the configured task on the method is ignored and the task (if any) associated with the imported unit of work is used. If the container begins a new unit of work when the method executes, then it is associated with the configured task name. Therefore, the last step in assembling applications for application profiling is to define a unit of work on any

method that has a task name (and eventually an Application Profile) associated with it. A unit of work can either be a transaction or an ActivitySession. See the topic, Defining container transactions for EJB modules, for a description on how to configure a transaction on an EJB module. The topic, Configuring transactional deployment attributes, describes how to define other transaction attributes. The topic, Using the ActivitySession service, describes how to use and create an ActivitySession unit of work. For more information about the relationships between tasks and units of work, see the topic, Tasks and units of work considerations.

What to do next

To complete the following tasks using assembly tools see the assembly tool documentation. The following tasks can be done using assembly tools:

- Automatic configuration of application profiling

  The assembly tool includes a static analysis engine that can assist you in configuring application profiling. The tool examines the compiled classes and the deployment descriptor of a Java EE application to determine the entry point of transactions, calculate the set of entities enlisted in each transaction, and determine whether the entities are read or updated during the course of each identified transaction.
- Automatically configure application profiles and tasks.

  Automatically configure application profiling for an application through static analysis.
- Apply profile-scoped access intent policies to entity beans.

  Configure entities with access intent for an application profile.
- Create a custom access intent policy.

  Define a custom access intent policy, which can be configured for Enterprise JavaBeans (EJB) 2.x and 3.0 entity beans.
- Create an application profile.

  An application profile contains a set of access intent policies applied to an application's entity beans. The access intent policies are only applied for requests that are associated with tasks configured on the application profile.
- Configure container-managed tasks for application clients.

  For application clients that programmatically begin either a transaction or ActivitySession only, you must configure an application client's container-managed task to associate requests from the client with an application profile.
- Configure container-managed tasks for Web components.

  For Web components that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession only, you can configure Web components application-managed tasks to associate requests from a servlet or JavaServer Pages (JSP) file with application profiles.
- Configure container-managed tasks for Enterprise JavaBeans.

  For methods that cause a new transaction or ActivitySession to be started either by the container or programmatically by the EJB developer, you can configure an enterprise bean's container-managed tasks to associate requests from the bean with application profiles.
- Configure container-managed tasks for application clients.

  For application clients that programmatically begin either a transaction or ActivitySession only, you must configure an application client's container-managed task to associate requests from the client with an application profile.
- Configure application-managed tasks for Web components.

  For Web components that programmatically begin either a transaction or ActivitySession only, you can configure a Web component's container-managed task to associate requests from a servlet or JSP file with an application profile.
- Configure application-managed tasks for Enterprise JavaBeans.

For Enterprise JavaBeans that programmatically set the configured task and then programmatically begin either a transaction or ActivitySession only, you can configure EJB application-managed tasks to associate requests from the bean with application profiles.

Chapter 3. Developing Asynchronous beans

This page provides a starting point for finding information about asynchronous beans.

Asynchronous beans and asynchronous scheduling facilities offer performance enhancements for resource-intensive tasks by enabling single tasks to run as multiple tasks.

Developing work objects, event listeners, and asynchronous scopes

Developing work objects to run code in parallel

You can run work objects in parallel, or in a different Java Platform, Enterprise Edition (Java EE) context, by wrapping the code in a work object.

Before you begin

Your administrator must have configured at least one work manager using the administrative console.

About this task

To run code in parallel, wrap the code in a work object.

Procedure

1. Create a work object.

   A work object implements the com.ibm.websphere.asynchbeans.Work interface. For example, you can create a work object that dynamically subscribes to a topic and any component that has access to the event source can add an event on demand:

```
class SampleWork implements Work
{
 boolean released;
 Topic targetTopic;
 EventSource es;
 TopicConnectionFactory tcf;

 public SampleWork(TopicConnectionFactory tcf, EventSource es, Topic targetTopic)
 {
  released = false;
  this.targetTopic = targetTopic;
  this.es = es;
  this.tcf = tcf;
 }

 synchronized boolean getReleased()
 {
  return released;
 }

 public void run()
 {
  try
  {
   // setup our JMS stuff.
   TopicConnection tc = tcf.createConnection();
   TopicSession sess = tc.createSession(false, Session.AUTOACK);
   tc.start();

   MessageListener proxy = es.getEventTrigger(MessageListener.class, false);
   while(!getReleased())
   {
```

```
    // block for up to 5 seconds.
    Message msg = sess.receiveMessage(5000);
    if(msg != null)
    {
     // fire an event when we get a message
     proxy.onMessage(msg);
    }
   }
   tc.close();
  }
  catch (JMSException ex)
  {
   // handle the exception here
   throw ex;
  }
  finally
  {
   if (tc != null)
   {
    try
    {
     tc.close();
    }
    catch (JMSExceptin ex1)
    {
     // handle exception
    }
   }
  }
 }

 // called when we want to stop the Work object.
 public synchronized void release()
 {
  released = true;
 }
}
```

As a result, any component that has access to the event source can add an event on demand, which allows components to subscribe to a topic in a more scalable way than by simply giving each client subscriber its own thread. The previous example is fully explored in the WebSphere Trader Sample. Refer to the Samples section of the Information Center for details.

2. Determine the number of work managers needed by this application component.
3. Look up the work manager or managers using the work manager resource reference (or logical name) in the java:comp namespace. (For more information on resource references, refer to the References topic.)

   ```
   InitialContext ic = new InitialContext();
   WorkManager wm = (WorkManager)ic.lookup("java:comp/env/wm/myWorkManager");
   ```

   The resource reference for the work manager (in this case, `wm/myWorkManager`) must be declared as a resource reference in the application deployment descriptor.
4. Call the WorkManager.startWork() method using the work object as a parameter. For example:

   ```
   Work w = new MyWork(...);
   WorkItem wi = wm.startWork(w);
   ```

   The startWork() method can take a startTimeout parameter. This specifies a hard time limit for the Work object to be started. The startWork() method returns a work item object. This object is a handle that provides a link from the component to the now running work object.
5. [Optional] If your application component needs to wait for one or more of its running work objects to complete, call the WorkManager.join() method. For example:

```
WorkItem wiA = wm.start(workA);
WorkItem wiB = wm.start(workB);
ArrayList l = new ArrayList();
l.add(wiA);
l.add(wiB);
if(wm.join(l, wm.JOIN_AND, 5000)) // block for up to 5 seconds
{

// both wiA and wiB finished
}
else
{

// timeout

// we can check wiA.getStatus or wiB.getStatus to see which, if any, finished.
}
```

This method takes an array list of work items which your component wants to wait on and a flag that indicates whether the component will wait for one or all of the work objects to complete. You also can specify a timeout value.

6. Use the release() method to signal the unit of work to stop running. The unit of work then attempts to stop running as soon as possible. Typically, this action is completed by toggling a flag using a thread-safe approach like the following example:

```
public synchronized void release()
{
 released = true;
}
```

The Work.run() method can periodically examine this variable to check whether the loop exits or not.

Work objects

A work object is a type of asynchronous bean used by application components to run code in parallel or in a different Java Platform, Enterprise Edition (Java EE) context.

A work object implements the com.ibm.websphere.asynchbeans.Work interface. A work object is essentially a java.lang.Runnable object that is serializable and provides additional methods. For details, refer to the Interface Work in the generated API documentation.

A component wanting to run work in parallel, or in a different Java EE context, locates a work manager in Java™ Naming and Directory Interface (JNDI), then calls the WorkManager.startWork() method using the work object as a parameter.

The startWork() method returns a work item object. This object is a handle that provides a link from the component to the now running work object. The work item object is typically used when the component needs to wait for one or more of its running work objects to complete. The WorkManager.join() method takes an array list of work items that the component wants to wait on, and a flag indicating whether the component will wait for all or one of the work objects to complete. A timeout can be specified, which prevents the component from waiting indefinitely.

The application does not create Java SE Development Kit 6 (JDK 6) threads because they are not managed threads. Plus, these threads are not affiliated with the Java EE environment, which makes them useless inside an application server. In addition, these threads have no Java EE context (for example, a java:comp) and are not authenticated when they fire. Work object threads are fully supported by the application server and have the same properties as other asynchronous beans.

Example: Creating work objects

You can create a work object that dynamically subscribes to a topic and any component that has access to the event source can add an event on demand.

The following is an example of a work object that dynamically subscribes to a topic:

```
class SampleWork implements Work
{
 boolean released;
 Topic targetTopic;
 EventSource es;
 TopicConnectionFactory tcf;

 public SampleWork(TopicConnectionFactory tcf, EventSource es, Topic targetTopic)
 {
  released = false;
  this.targetTopic = targetTopic;
  this.es = es;
  this.tcf = tcf;
 }

 synchronized boolean getReleased()
 {
  return released;
 }

 public void run()
 {
  try
  {
   // setup our JMS stuff.
   TopicConnection tc = tcf.createConnection();
   TopicSession sess = tc.createSession(false, Session.AUTOACK);
   tc.start();

   MessageListener proxy = es.getEventTrigger(MessageListener.class, false);
   while(!getReleased())
   {
    // block for up to 5 seconds.
    Message msg = sess.receiveMessage(5000);
    if(msg != null)
    {
     // fire an event when we get a message
     proxy.onMessage(msg);
    }
   }
   tc.close();
  }
  catch (JMSException ex)
  {
   // handle the exception here
   throw ex;
  }
  finally
  {
   if (tc != null)
   {
    try
    {
     tc.close();
    }
    catch (JMSExceptin ex1)
    {
     // handle exception
    }
   }
  }
 }

 // called when we want to stop the Work object.
 public synchronized void release()
```

```
 {
  released = true;
 }
}
```

As a result, any component that has access to the event source can add an event on demand, which allows components to subscribe to a topic in a more scalable way than by simply giving each client subscriber its own thread. The previous example is fully explored in the WebSphere Trader Sample. See the Samples section of the Information Center for details.

Developing event listeners

Application components that listen for events can use the EventSource.addListener() method to register an event listener object (a type of asynchronous bean) with the event source to which the events will be published. An event source also can fire events in a type-safe manner using any interface.

About this task

Notifications between components within a single EAR file are handled by a special event source. See the Using the application notification service topic for more information about notifications.

Procedure

1. Create an event listener object, which can be any type. For example, see the following interface code:

```
interface SampleEventGroup
{
  void finished(String message);
}

class myListener implements SampleEventGroup
{
  public void finished(String message)
  {
  // This will be called when we 'finish'.
  }
}
```

2. Register the event listener object with the event source. For example, see the following code:

```
InitialContext ic = ...;
EventSource es = (EventSource)ic.lookup("java:comp/websphere/ApplicationNotificationService");
myListener l = new myListener();
es.addListener(l);
```

   This enables the myListener.finished() method to be called whenever the event is fired. The following code example shows how this event might be fired:

```
InitialContext ic = ...;
EventSource es = (EventSource)ic.lookup("java:comp/websphere/ApplicationNotificationService");
myListener proxy = es.getEventTrigger(myListener.class);
// fire the 'event' by calling the method
// representing the event on the proxy
proxy.finished("done");
```

Example

You can fire a listenerCountChanged event that produces a proxy for the interface on which the method fires. Calling the method corresponding to the event on the proxy implements the EventSourceEvents interface. The same proxy can be used to send multiple events simultaneously.

The following code example demonstrates how to fire a listenerCountChanged event:

```
// Imagine this snippet inside an EJB or servlet method.
// Make an inner class implementing the required event interfaces.
EventSourceEvents listener = new Object() implements EventSourceEvents.class
{
 void listenerCountChanged(EventSource es, int old, int newCount)
```

```
 {
  try
  {
     InitialContext ic = new InitialContext();
   // Here, the asynchronous bean can access an environment variable of
   // the component which created it.
    int i = (Integer)ic.lookup("java:comp/env/countValue").intValue());
   if(newCount == i)
   {
    // do something interesting
   }
   // call this event when the following code executes:
  }
  catch(NamingException e)
  {
  }
 }
 void listenerExceptionThrown( EventSource es, Object listener,
       String methodName, Throwable exception)
 {
 }
 void unexpectedException(EventSource es, Object runnable, Throwable exception)
 {
 }
}
// register it.
es.addListener(listener);

...

// now fire an event which the previous listener receives.
EventSourceEvents proxy = (EventSourceEvents)
   es.getEventTrigger(EventSourceEvents.class, false);

proxy.listenerCountChanged(es, 0, 1);

// now, fire another event, you can call any of the methods.
proxy.listenerCountChanged(es, 4, 5);
```

The output in this example is a proxy for the interface on which the method fires. Then, call the method corresponding to the event on the proxy. This action causes the same method with the same parameters to be called on any event listeners that implement the EventSourceEvents interface and that were previously registered with the EventSource "es". The same proxy can be used to send multiple events simultaneously.

The boolean parameter on the getEventTrigger() method is `sameTransaction`. When the `sameTransaction` parameter is `false`, a new transaction is started for each event listener invoked and these event listeners can be called in parallel to the caller. However, the event() method is blocked until all of the event listeners are notified. If the `sameTransaction` parameter is `true`, then the current transaction (if any) on the thread is used for all of the event listeners. The event listeners share the transaction of the method that fired the event. For that reason, all event listeners must run serially in an undetermined order. The order that listeners are called is undefined, and the order in which listeners are registered does not act as a guide for the order used at run time. The method on the proxy does not return until all of the event listeners are called, which means that this action is a synchronous operation.

The parameters that references and listeners pass do not interfere with the function of these references, unless you configure the method to do so. For example, event listeners can be used as collaborators and add data to a map, which was a parameter. Each event listener runs on its own transaction, independent of any transaction that is active on the thread. Extreme care must be taken when the `sameTransaction` parameter is `false` because the parameters can be accessed by multiple threads.

Using the application notification service

During the application lifetime, individual J2EE components (servlets or enterprise beans) within a single EAR file might need to signal each other. There is an event source in the java:comp namespace that is bound into all components within an EAR file that can be used for notification.

About this task

The JNDI name for this event source, in the java:comp namespace that is bound into all components within an EAR file, is:

```
java:comp/websphere/ApplicationNotificationService
```

Components within the same application can fire asynchronous events and register event listeners using this application notification service. Startup beans can be used to register these event listeners at application startup or they can be registered dynamically at run time.

Procedure

To have your enterprise bean or servlet use the application notification service, write code similar to the following example:

```
InitialContext ic = new InitialContext();
EventSource appES = (EventSource)
  ic.lookup("java:comp/websphere/ApplicationNotificationService");
// now, the application can add a listener using the EventSource.addListener method.
// MyEventType is an interface.
MyEventType myListener = ...;
AppES.addListener(myListener);

// later another component can fire events as follows
InitialContext ic = new InitialContext();
EventSource appES = (EventSource)
ic.lookup("java:comp/websphere/ApplicationNotificationService");

// This highlights a constant string on the EventSource interface which
// specifies the 'java:comp/websphere/ApplicationNotificationService' string.
ic.lookup(appES.APPLICATION_NOTIFICATION_EVENT_SOURCE)
// now, the application can add a listener using the EventSource.addListener method.
MyEventType proxy = appES.getEventTrigger(MyEventType.class, false);
proxy.someEvent(someArguments);
```

Example

Example: Firing a listenerCountChanged event

You can fire a listenerCountChanged event that produces a proxy for the interface on which the method fires. Calling the method corresponding to the event on the proxy implements the EventSourceEvents interface. The same proxy can be used to send multiple events simultaneously.

The following code example demonstrates how to fire a listenerCountChanged event:

```
// imagine this snippet inside an EJB or servlet method.
// Make an inner class implementing the required event interfaces.
EventSourceEvents listener = new Object() implements EventSourceEvents.class
{
 void listenerCountChanged(EventSource es, int old, int newCount)
 {
  try
  {
     InitialContext ic = new InitialContext();
   // Here, the asynchronous bean can access an environment variable of
   // the component which created it.
    int i = (Integer)ic.lookup("java:comp/env/countValue").intValue());
   if(newCount == i)
   {
```

```
    // do something interesting
   }
   // call this event when the following code executes:
  }
  catch(NamingException e)
  {
  }
 }
 void listenerExceptionThrown( EventSource es, Object listener,
       String methodName, Throwable exception)
 {
 }
 void unexpectedException(EventSource es, Object runnable, Throwable exception)
 {
 }
}
// register it.
es.addListener(listener);

...

// now fire an event which the previous listener receives.
EventSourceEvents proxy = (EventSourceEvents)
   es.getEventTrigger(EventSourceEvents.class, false);

proxy.listenerCountChanged(es, 0, 1);

// now, fire another event, you can call any of the methods.
proxy.listenerCountChanged(es, 4, 5);
```

The output in this example is a proxy for the interface on which the method fires. Then, call the method corresponding to the event on the proxy. This action causes the same method with the same parameters to be called on any event listeners that implement the EventSourceEvents interface and that were previously registered with the EventSource "es". The same proxy can be used to send multiple events simultaneously.

The boolean parameter on the getEventTrigger() method is `sameTransaction`. When the `sameTransaction` parameter is `false`, a new transaction is started for each event listener invoked and these event listeners can be called in parallel to the caller. However, the event() method is blocked until all of the event listeners are notified. If the `sameTransaction` parameter is `true`, then the current transaction (if any) on the thread is used for all of the event listeners. The event listeners share the transaction of the method that fired the event. For that reason, all event listeners must run serially in an undetermined order. The order that listeners are called is undefined, and the order in which listeners are registered does not act as a guide for the order used at run time. The method on the proxy does not return until all of the event listeners are called, which means that this action is a synchronous operation.

The parameters that references and listeners pass do not interfere with the function of these references, unless you configure the method to do so. For example, event listeners can be used as collaborators and add data to a map, which was a parameter. Each event listener runs on its own transaction, independent of any transaction that is active on the thread. Extreme care must be taken when the sameTransaction parameter is `false` because the parameters can be accessed by multiple threads.

Developing asynchronous scopes

`Asynchronous scopes` are units of scoping that comprise a set of alarms, subsystem monitors, and child asynchronous scopes. You can create asynchronous scopes, starting with the parent.

About this task

Using asynchronous scopes can involve some or all of the following steps:

Procedure

1. Create asynchronous scopes. Create the parent asynchronous scope object by using a unique parameter name that calls the AsynchScopeManager.createAsynchScope() method. You can store properties in an asynchronous scope object. This storage provides Java 2 Enterprise Edition (J2EE) applications with a way to store a non-serializable state that otherwise cannot be stored in a session bean. You also can create child asynchronous scopes, which is useful for scoping data beneath the parent.
2. Listen for alarm notifications
   a. Create a listener object by implementing the AlarmListener interface. For more information, refer to the AlarmListener interface in the generated API documentation.
   b. Supply this object to the AlarmManager.create() method, as the target for the alarm. The create() method takes the following parameters:
      Target for the alarm
      The target on which the fired() method is called when the alarm is fired.
      Context
      The context object for the alarm. This object is useful for supplying alarm-specific data to the listener and supports a single listener for multiple alarms.
      Interval
      The number of milliseconds before the alarm fires.

      After the specified interval, the alarm fires and the fired() method of the listener is called with the firing alarm as a parameter. The alarm object is returned. By calling methods on this object, you can cancel or reschedule the alarm.
3. Monitor remote systems.
   a. Implement a mechanism for detecting messages sent from the remote system. For example, publish and subscribe messaging.
   b. Create a subsystem manager object by calling the SubsystemMonitorManager.create() method with the following parameters:
      Name Each subsystem monitor must have a unique name.
      Heartbeat interval
      The expected interval, in milliseconds, between heartbeats.
      Missed heart beats until stale or suspect
      The number of heartbeats that can be missed before the subsystem is marked as stale.
      Missed heart beats until dead
      The number of heartbeats that can be missed before the system is marked as dead.
   c. Create an object that implements the SubsystemMonitorEvents interface. For more information, see the SubsystemMonitorEvents in the generated API documentation.
   d. Add an instance of this object to the subsystem monitor using the SubsystemMonitor.addListener() method.
   e. Whenever a heartbeat message arrives from the remote system, call the SubsystemMonitor ping() method.

   The subsystem monitor configures alarms to track the heartbeat status of the remote system. When the ping() method is called, the alarms are reset. If an alarm fires, the ping() method is not called; that is, the application did not receive a heartbeat from the monitored subsystem.

Example

Asynchronous scopes are useful in stateful server applications. An application can have a startup bean that creates an asynchronous scope on a named work manager. The application also might create subsystem monitors to monitor the health of any remote systems on which the application is dependent.

When a client attaches to the server, the application creates a child asynchronous scope that is owned by the application asynchronous scope for the client and named using the client ID. A subsystem monitor for

monitoring the client might be created on the client asynchronous scope. If the client times out, a callback can clean up the client state on the server. Callbacks can be attached to the application subsystem monitors, on behalf of the client. When a remote system becomes unavailable, the client code in the server is notified and an event is sent to the client to warn that a critical remote system has failed. For example, the failure might be a data feed in an electronic trading application.

Asynchronous scopes

An asynchronous scope (AsynchScope object) is a unit of scoping provided for use with asynchronous beans.

Asynchronous scopes are collections of alarms, subsystem monitors, and child asynchronous scopes that enable a relationship to form. Each asynchronous scope uses a single work manager.

Each AsynchScope object owns and controls the life cycle of the following objects:

Child asynchronous scopes
: Each AsynchScope object extends the AsynchScopeManager interface, which is a factory for AsynchScope objects. (For more information on the AsynchScopeManager interface, refer to the generated API documentation). Any asynchronous scope can therefore create named asynchronous scopes (children). Child asynchronous scopes can be useful for scoping data underneath the parent. All of the child asynchronous scopes must be uniquely named. These children are destroyed if the parent asynchronous scope is destroyed.

Alarms
: Each asynchronous scope has an associated alarm manager. All of the alarms created by the alarm manager are automatically cancelled if the associated asynchronous scope is destroyed.

Subsystem monitors
: Each asynchronous scope has a subsystem monitor manager, which manages a set of subsystem monitors associated with the asynchronous scope. When the asynchronous scope is destroyed, all of the associated subsystem monitors also are destroyed.

In summary, asynchronous scopes can be organized into an acyclic tree. The life cycle of each asynchronous scope is directly coupled to that of its parent asynchronous scope. Each asynchronous scope is associated with a set of alarms and subsystem monitors, and an optional set of child asynchronous scopes. These objects are cancelled and destroyed when the asynchronous scope is destroyed.

Asynchronous scope state

Each asynchronous scope has an associated map, in which applications can store their state in the form of name and value pairs.

Asynchronous scope events

Each asynchronous scope is also an event source. Applications can therefore register event listeners against the asynchronous scope. The event listeners can receive notification if, for example, the AsynchScope object is about to be destroyed.

Applications also can use this event source to fire events only to listeners of this asynchronous scope. For example, an AsynchScope object created for a client session might be used to fire asynchronous events to parties interested in that client.

Alarms

An alarm runs Java Platform, Enterprise Edition (Java EE) context-aware code at a given time interval. Alarm objects are fine-grained, nonpersistent, transient, and can fire at millisecond intervals.

Alarms are run using a thread pool associated with the work manager that owns the associated asynchronous scope. You must create a work manager instance to create an alarm. Refer to the Configuring work managers topic for more information.

The AlarmManager.createAlarm() method takes an application-written object that implements the AlarmListener interface. For more information on the AlarmListener interface, refer to the generated API documentation. The fired method is called when the alarm expires. The createAlarm() method returns a non-serializable handle, which can be used to cancel or reset the alarm. All of the pending alarms are cancelled when its associated AsynchScope object is destroyed.

best-practices: The Java SE Development Kit 6 (JDK6)already has a timer mechanism, so why create a new one? The JDK 6 is a Java Platform, Standard Edition (Java SE) feature that knows nothing about the Java EE environment. Timers fired by the Java SE feature do not run on a managed thread and are therefore unusable inside an application server. These timers also lack a Java EE context (that is, a java:comp value) and are not authenticated when they fire. The asynchronous scope alarms are fully supported by the product and have the same properties as any other asynchronous bean.

Alarm performance

The alarm subsystem is designed to handle a large number of alarms. However, do not expect alarms to process heavy loads when they are firing because this activity slows the processing of later alarms. If an alarm needs to process a heavy load, design a work object that is activated by a work manager. This procedure moves the heavy processing to a different thread and enables the alarm threads to process alarms unhampered. All of the alarms owned by asynchronous scopes that are owned by a single work manager share a common thread pool. The properties of this thread pool can be tuned at the work manager level using the administrative console.

Subsystem monitors

A subsystem monitor is an object that monitors the health of a remote system. It uses an event source to inform all registered listeners of the health of the system.

AdvancedJava Platform, Enterprise Edition (Java EE) applications often rely on remote, non-managed, non-Java EE systems. These remote systems can periodically send clients a message to indicate that they are working. A subsystem monitor is a set of alarms that tracks indicator messages or heart beats from a remote system.

An application creates a subsystem monitor by calling the SubsystemMonitorManager.create() method with the following parameters:

Name Each subsystem monitor must be uniquely named.

Heart beat interval
The time period, in milliseconds, between arriving heart beat messages.

Missed heart beats until stale or suspect
The number of heart beats that can be missed before the subsystem is marked as stale. This designation indicates that the subsystem might be having problems.

Missed heart beats until dead
The number of heart beats that can be missed before the system is considered down. The system then is marked as dead.

The subsystem monitor configures alarms to track the heart beat status. Whenever the ping() method is called, the alarms are reset. If an alarm fires, the ping() method has not been called; that is, the application did not receive a heart beat from the monitored subsystem. When the number of **Missed heart beats until stale** value has elapsed without a ping, a stale event is fired. Later, if the number of **Missed heart beats until dead** value elapses without a ping, a dead event is fired. If a ping is received after a stale or dead notification, a fresh event is sent, which indicates that the subsystem is alive again.

Make the **Missed heart beats until dead** value greater or equal to the **Missed heart beats until stale** value. If **Missed heart beats until stale** value equals the **Missed heart beats until dead** value, then a stale event is not published. Only a dead event is published.

You can register a listener that implements the SubsystemMonitorEvents interface for applications that require notification of events. For more information on the SybsystemMonitorEvents interface, refer to the generated API documentation.

Heart beat messages can be transmitted using a variety of mechanisms. The application must call the SubsystemMonitor ping() method whenever a heart beat message arrives from a remote system, but the method used to detect these messages is up to the application. For example, you might use a Java Message Service (JMS) publish or subscribe implementation or even a third-party Java messaging product that does not implement JMS.

Asynchronous scopes: Dynamic message bean scenario

Java Platform, Enterprise Edition (Java EE) now supports message-driven beans, but the beans are static. This scenario provides information about how to set up the environment to enable the dynamic message bean.

All of the message sources must be known in advance and bound at deployment time. This action is not always viable, especially in fluid messaging environments such as those found in brokerages. Some environments have publish-subscribe topic spaces that are continually changing and clients need servers that can subscribe on demand to an arbitrary topic.

An asynchronous bean application can create a work object that performs a blocking receive on a Java Message Service (JMS) topic and then publishes the message as an event on an application-defined event source. Clients requiring a subscription to that message can add an event listener to the event source. The event source can inform the work object when there are no listeners. Then, the event source can shut down and make the JMS and thread resources available. The work object registers a listener with its own event source. When the count is one again, the work object knows that it is the only listener and it is time to shut down the work object. The WebSphere Trader Sample uses this pattern to dynamically subscribe to JMS topics at run time to gather stock prices. For more information, see the overview of the samples.

How does the server catch clients that disconnect or crash? It creates a subsystem monitor to watch the client and adds an event listener to catch dead events. When a dead event occurs, the server application can clean up the client server state. For example, the server application can remove the client event listener from the dynamic message bean, thereby allowing the server to subscribe to a dynamic topic only when it is needed.

Assembling timer and work managers

Assembling applications that use work managers and timer managers

The work manager and timer manager objects are both supported for assembling applications that implement the asynchronous bean technology. You can assemble either work managers or time managers.

Before you begin

Configure at least one work manager or timer manager using the administrative console.

About this task

Complete the steps to either assemble work managers or time managers.

Procedure

1. Assemble applications that use asynchronous beans work managers.
2. Assemble applications that use CommonJ work managers.
3. Assemble applications that use CommonJ timer managers.

Assembling applications that use a CommonJ WorkManager

When a work manager has been configured, if it references a logical work manager it must be bound to a physical work manager using an assembly tool. Then resources can be created and bound to a physical work manager.

Before you begin

Your administrator needs to configure at least one work manager using the administrative console.

About this task

If your application references one or more logical work managers, the logical work managers must be bound to one or more physical work managers using an assembly tool, such as Rational Web Developer.

Procedure

1. Declare a resource reference for each work manager (required action by the application developer). This forms an EAR file. (For more information on resource references, refer to the References. topic)
2. Bind each resource reference to a physical work manager, using an assembly tool, such as Rational Web Developer.
3. Add a resource reference with the type commonj.work.WorkManager to the application deployment descriptor. The application can look up this work manager using its resource reference name in java:comp. Now, you can use an assembly tool or Rational Application Developer to specify which resource references are bound to the physical commonj.work.WorkManager.

   Attention: The previous steps outline the same process used for data sources.

Assembling applications that use timer managers

When a work manager has been configured, if it references a logical work manager it must be bound to a physical work manager using an assembly tool. Then resources can be created and bound to a physical timer.

Before you begin

Your administrator needs to configure at least one timer manager using the administrative console.

About this task

If your application references one or more logical timer managers, the logical timer managers must be bound to one or more physical timer managers using an assembly tool, such as the Rational Application Developer.

Procedure

1. Declare a resource reference for each timer manager (required action by the application developer). This forms an EAR file. (For more information on resource references, refer to the References topic.)
2. Bind each resource reference to a physical timer manager, using an assembly tool.
3. Add a resource reference with the type commonj.timers.TimerManager to the application deployment descriptor. The application then can look up this timer manager using its resource reference name in java:comp. The assembly tool can specify which resource references are bound to a physical timer manager.

Attention: The previous steps outline the same process used for data sources.

Assembling applications that use asynchronous beans work managers

When a work manager has been configured, if it references a logical work manager it must be bound to a physical work manager using an assembly tool. Then resources can be created and bound to a physical work managers.

Before you begin

Your administrator needs to configure at least one work manager using the administrative console.

About this task

If your application references one or more logical work managers, the logical work managers must be bound to one or more physical work managers using an assembly tool.

The CommonJ 1.1 interfaces are supported. Both asynchronous beans and CommonJ interfaces can use one configuration work manager object. The type of interface implemented is resolved during the JNDI lookup time. The type of interface used is determined by the one specified in the resource-reference, instead of the one specified in the configuration object. So, there can be one resource-reference for each interface, per configuration object. Each resource-reference lookup returns the appropriate type of instance. For example, there are two resource-references defined for the `wm/MyWorkManager:` `wm/ABWorkMgr` and `wm/CommonJWorkMgr`. The WebSphere Application Server run time returns the correct interface for each resource-reference lookup.

Procedure

1. Declare a resource reference for each work manager (required action by the application developer). This action results in an EAR file. For more information on resource references, refer to the References topic.
2. Use an assembly tool to bind each resource reference to a physical work manager.
3. Add a resource reference with the type com.ibm.websphere.asynchbeans.WorkManager to the application deployment descriptor. The application then can look up this work manager using its resource reference name in java:comp. The assembly tool or Rational Application Developer then can specify which resource references are bound to a physical work manager.

   Attention: Use the same previous steps to configure data sources.

Chapter 4. Developing applications that use the Bean Validation API

The Bean Validation API is introduced with the Java Enterprise Edition 6 platform as a standard mechanism to validate Enterprise JavaBeans in all layers of an application, including, presentation, business and data access. Before the Bean Validation specification, the JavaBeans were validated in each layer. To prevent the reimplementation of validations at each layer, developers bundled validations directly into their classes or copied validation code, which was often cluttered. Having one implementation that is common to all layers of the application simplifies the developers work and saves time.

Bean Validation

The Bean Validation API is introduced with the Java Enterprise Edition 6 platform as a standard mechanism to validate JavaBeans in all layers of an application, including presentation, business, and data access.

Before the Bean Validation specification, JavaBeans were validated in each layer. To prevent the re-implementation of validations at each layer, developers bundled validations directly into their classes or copied validation code, which was often cluttered. Having one implementation that is common to all layers of the application simplifies the developers work and saves time.

The Bean Validation specification defines a metadata model and an API that are used to validate JavaBeans for data integrity. The metadata source is the constraint annotations defined that can be overridden and extended using XML validation descriptors. The set of APIs provides an ease of use programming model allowing any application layer to use the same set of validation constraints. Validation constraints are used to check the value of annotated fields, methods, and types to ensure that they adhere to the defined constraint.

Constraints can be built in or user-defined. Several built-in annotations are available in the javax.validation.constraints package. They are used to define regular constraint definitions and for composing constraints. For a list of built-in constraints, see the topic, “Bean validation built-in constraints” on page 32. For more details about the Bean Validation metadata model and APIs see the JSR 303 Bean Validation specification document.

The following example is a simple Enterprise JavaBeans (EJB) class that is decorated with built-in constraint annotations.

```
public class Home  {
   @Size(Max=20)
    String builder;
    @NotNull @Size(Max=20)
    String address;

    public String getAddress() {
         return address;
    }

    public String getBuilder() {
         return address;
    }
    public String setAddress(String newAddress) {
         return address = newAddress;
    }

    public String setBuilder(String newBuilder) {
         return builder = newBuilder;
    }
}
```

The @Size annotations on builder and address specify that the string value assigned should not be greater 20 characters. The @NotNull annotation on address indicates that it cannot be null. When the Home object is validated, the builder and address values are passed to the validator class defined for the @Size annotation. The address value is also be passed to the @NotNull validator class. The validator classes handle checking the values for the proper constraints and if any constraint fails validation, a ConstraintViolation object is created, and is returned in a set to the caller validating the Home object.

Validation APIs

The javax.validation package contains the bean validation APIs that describe how to programmatically validate JavaBeans.

ConstraintViolation is the class describing a single constraint failure. A set of ConstraintViolation classes is returned for an object validation. The constraint violation also exposes a human readable message describing the violation.

ValidationException are raised if a failure happens during validation.

The Validator interface is the main validation API and a Validator instance is the object that is able to validate the values of the Java object fields, methods, and types. The bootstrapping API is the mechanism used to get access to a ValidatorFactory that is used to create a Validator instance. For applications deployed on the product, bootstrapping is done automatically. There are two ways for applications to get the validator or the ValidatorFactory. One way is injection, for example, using the @Resource annotation, and the other way is the java: lookup.

The following example uses injection to obtain a ValidatorFactory and a Validator:

```
@Resource ValidatorFactory _validatorFactory;
@Resource Validator _validator;
```

Attention: When using @Resource to obtain a Validator or ValidatorFactory, the authenticationType and shareable elements must not be specified.

The following example uses JNDI to obtain a ValidatorFactory and a Validator:

```
ValidatorFactory validatorFactory = (ValidatorFactory)context.lookup("java:comp/ValidatorFactory");
Validator validator = (Validator)context.lookup("java:comp/Validator");
```

Constraint metadata request APIs

The metadata APIs support tool providers, provides integration with other frameworks, libraries, and Java Platform, Enterprise Edition technologies. The metadata repository of object constraints is accessed through the Validator instance of a given class.

XML deployment descriptors

Besides declaring constraints in annotations, support exists for using XML to declare your constraints.

The validation XML description is composed of two kinds of xml files. The META-INF/validation.xml file describes the bean validation configuration for the module. The other XML file type describes constraints declarations and closely matches the annotations declaration method. By default, all constraint declarations expressed through annotations are ignored for classes described in XML. It is possible to force validation to use both the annotations and the XML constraint declarations by using the ignore-annotation="false" setting on the bean. The product ensures that application modules deployed containing a validation.xml file and constraints defined in XML files are isolated from other module validation.xml and constraint files by creating validator instances specific to the module containing the XML descriptors.

Advanced bean validation concepts

The Bean Validation API provides a set of built-in constraints and an interface that enables you to declare custom constraints. This is accomplished by creating constraint annotations and declaring an annotation on a bean type, field, or property. Composing constraints is also done by declaring the constant on another constraint definition.

The following example shows creating a CommentChecker constraint that is defined to ensure a comment string field is not null. The comment text is enclosed by brackets, such as [*text*].

```
package com.my.company;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import javax.validation.Constraint;
import javax.validation.Payload;

@Documented
@Constraint(validatedBy = CommentValidator.class)
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
public @interface CommentChecker {
       String message() default "The comment is not valid.";
       Class<?>[] groups() default {};
       Class<? extends Payload>[] payload() default {};
      ...}
```

The next example shows the constraint validator that handles validating elements with the @CommentChecker annotation. The constraint validator implements the ConstraintValidator interface provided by the Bean Validation API.

```
package com.my.company;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class CommentValidator implements ConstraintValidator<CommentChecker, String> {
    public void initialize(CommentChecker arg0) {
    }
    public boolean isValid(String comment, ConstraintValidatorContext context) {
        if (comment == null) {
            // Null comment is not allowed, fail the constraint.
            return false;
        }
        if (!comment.contains("[") && !comment.contains("]")) {
            // Can't find any open or close brackets, fail the constraint
            return false;
        }
        // Ignore leading and trailing spaces
        String trimmedComment = comment.trim();
        return // validate '[' prefix condition
               trimmedComment.charAt(0) == '[' &&
               // validate ']' suffix condition
               trimmedComment.charAt(trimmedComment.length()!-1) == ']';
    }
}
```

After the @CommentChecker is defined, it can be used to ensure that the comment string field is a valid comment based on the CommentValidator isValid() implementation. The following example shows the use of the @CommentChecker constraint. When the myChecker bean is validated, the comment string is validated by the CommentValidator class to ensure the constraints defined are met.

```
package com.my.company;
public myChecker {

    @CommentChecker
    String comment = null;
    ...
}
```

The product provides a specific bean validation provider, but it might be necessary for an application to use or require another provider.

This method can be accomplished by using the validator methods to set the provider programmatically and create a validation factory. Or, by using the validation.xml default-provider element. The specific provider that is defined and used to create the validation factory and not the default provider provided by the application server in the default implementation. If you want to ensure that the user-provided implementation does not conflict with the default implementation, the server or application class loading parameter, the class loader order should be set to be loaded with local class loader first (parent last). See additional information in the class loading documentation on how to set this setting.

The Bean Validation specification indicates that if more than one validation.xml file is found in the class path, a ValidationException occurs. However, WebSphere Application Server supports an environment where multiple teams develop modules that are assembled and deployed into the Application Server together. In this environment, all EJB modules within an application are loaded with the same class loader and it is possible to configure the application class loaders so that all EJB and web archive (WAR) modules are loaded by a single class loader. Because of this, the product provides support for multiple validation.xml files in the same class path.

When an application using bean validation and XML descriptors contains multiple EJB modules and web modules, each validation.xml file is associated with a validation factory that is specific to that module. In this environment, any constraint-mapping elements that are defined are only looked up in the module where the validation.xml file is defined. For example, if an EJB module building.jar contains a META-INF/validation.xml file and the validation.xml file defined the following constraints, both the META-INF/constraints-house.xml and META-INF/constraints-rooms.xml files must also be located in the building.jar file:

```
<constraint-mapping>META-INF/constraints-house.xml</constaint-mapping>
<constraint-mapping>META-INF/constraints-rooms.xml</constraint-mapping>
```

The exception to this behavior is when all bean validation constraints classes and configuration are visible to all application modules. In a case where a single validation.xml file is defined in an EAR file, and no other validation.xml files are visible in a module's class path, any module that creates a validator factory or validator will use the validation.xml file that is defined in the EAR file. This makes it possible for other modules to create a validator factory that uses the validation.xml file of another module, if the class path has been configured so that both modules are visible on the same class path and only one validation.xml file is visible to those modules.

For a more detailed understanding about the Bean Validation APIs and metadata see the JSR 303 Bean Validation specification document.

Bean validation built-in constraints

Use this information to look up information about Bean Validation API built-in constraints.

The Bean Validation API is supported by constraints that are primarily expressed through annotations. The constraints are added to a class, field, or method of an Enterprise JavaBeans (EJB) component. The annotated element value is checked by the constraint.

Constraints can be built in or user defined. Several built-in annotations are available in the javax.validation.constraints package. They are used to define regular constraint definitions and for composing constraints.

The following table is a list of constraints and usage.

Table 1. Bean validation built-in constraints. Use of bean validation built-in constraints

Constraint	Usage
@Null	Specifies that the configuration property decorated with this annotation must have a null value. This constraint accepts any type.
@NotNull	Specifies that the configuration property decorated with this annotation must not have a null value. That is, the property is required. This constraint accepts any type.
@AssertTrue	Specifies that the configuration property decorated with this annotation must be true. Supported value types are boolean and Boolean. Null elements are considered valid.
@AssertFalse	Specifies that the configuration property decorated with this annotation must be false. Supported value types are boolean and Boolean. Null elements are considered valid.
@Min	Specifies that the configuration property decorated with this annotation must have a value greater than or equal to the specified minimum. Supported value types are BigDecimal, BigInteger, byte, short, int, long and their respective wrappers. Null elements are considered valid.
@Max	Specifies that the configuration property decorated with this annotation must have a value less than or equal to the specified maximum. Supported value types are BigDecimal, BigInteger, byte, short, int, long and their respective wrappers. Null elements are considered valid.
@DecimalMin	Specifies that the configuration property decorated with this annotation must have a value higher or equal to the specified minimum. Supported value types are BigDecimal, BigInteger, String, byte, short, int, long and their respective wrappers. Null elements are considered valid.
@DecimalMax	Specifies that the configuration property decorated with this annotation must have a value lower or equal to the specified maximum. Supported value types are BigDecimal, BigInteger, String, byte, short, int, long and their respective wrappers. Null elements are considered valid.
@Size	Specifies that the configuration property decorated with this annotation must have a value between the specified boundaries (included). Supported value types are String (string length is evaluated), Collection (collection size is evaluated), Map (map size is evaluated), Array (array length is evaluated). Null elements are considered valid.
@Digits	Specifies that the configuration property decorated with this annotation must have a value within accepted range. Supported value types are BigDecimal, BigInteger, String, byte, short, int, long and their respective wrappers. Null elements are considered valid.
@Past	Specifies that the configuration property decorated with this annotation must have a date in the past. Now is defined as the current time according to the virtual machine. The calendar is used if the compared type is of type Calendar and the calendar is based on the current timezone and the current locale. Supported value types are java.util.Date, java.util.Calendar. Null elements are considered valid.
@Future	Specifies that the configuration property decorated with this annotation must have a date in the future. Now is defined as the current time according to the virtual machine. The calendar is used if the compared type is of type Calendar and the calendar is based on the current timezone and the current locale. Supported value types are java.util.Date, java.util.Calendar. Null elements are considered valid.
@Pattern	Specifies that the configuration property decorated with this annotation must match the following regular expression. The regular expression follows the Java regular expression conventions java.util.regex.Pattern. Supported type value is String. Null elements are considered valid.

Using bean validation in the product

The Java Enterprise Edition (Java EE) 6 specification includes the Bean Validation API that is a standard mechanism for validating JavaBeans in all layers of an application.

About this task

Before the Bean Validation specification, JavaBeans were validated in each layer. To prevent the reimplementation of validations at each layer, developers bundled validations directly into their classes or copied validation code, which was often cluttered. Having one implementation that is common to all layers of the application simplifies the developers work and saves time.

Bean validation is common to all layers of an application. Specifically, web applications have the following layers:

- Presentation

This layer represents how the user interacts with the application and might be built on a thin client or rich client.

- Business

  This layer coordinates the application, processes commands, makes logical decisions and evaluations and performs calculations. It also moves and processes data between the two other layers. The EJB contains business logic in WebSphere Application Server.

- Data access

  Your data is stored and retrieved from a database or file system at this layer. The business layer processes the data and sends it in usable form to the user interface. WebSphere Application Server supports several databases and methods of retrieving data. This layer also defines persistence.

For WebSphere Application Server, these layers are built and administered with several components in the product that are necessary for developing and deploying applications.

The product provides support for the Bean Validation API in the Java Platform, Enterprise Edition (Java EE) environment by providing a bean validation service in multiple Java EE technologies including Java Servlets, Enterprise JavaBeans, Java Persistence API (JPA) 2.0, Java EE Connector API (JCA) 1.6 and Java ServerFaces (JSF) 2.0. Bean validation provides these technologies a way to maintain data integrity in an integrated and standard environment.

Enterprise application development involves multiple teams developing numerous applications and modules that are assembled and deployed in an application server environment. The product ensures that each application and module data is validated independently. Validation is performed using only the constraints defined for the application and module.

What to do next

- Data access resources:

  Bean validation in RAR modules.

  The product validates resource adapter archive (RAR) Enterprise JavaBeans (EJB) constraints in compliance with the JCA version 1.6 specification. Resource adapters can use the built-in bean validation constraint annotations or provide a bean validation XML configuration to specify the validation requirements of resource adapter configuration properties to the application server.

- Using bean validation in JPA

  A new feature defined by the JPA 2.0 specification is the ability to seamlessly integrate with the Bean Validation API. With minimal effort, JPA 2.0 can be coupled with the validation provider for runtime data validation. By combining these two technologies, you get a standardized persistence solution with the added ability to perform standardized data validation.

- Using bean validation with JSF

  JSF previously was able do bean validation, but now it provides built-in support of the Bean Validation specification.

- Using bean validation in web container

  The web container provides an instance of ValidatorFactory and makes it available to JSF implementations by storing it in a servlet context attribute named javax.faces.validator.beanValidator.ValidatorFactory.

- Using bean validation with the embeddable container.

  To use bean validation with the embeddable EJB container, the javax.validation classes must exist in the class path. That can be done in one of two ways:

  – Include the JPA thin client that is located in the directory `${WAS_INSTALL_ROOT}\runtimes\com.ibm.ws.jpa.thinclient_8.0.0.jar` in the class path. See the topic, Running an embeddable container, and the information about JPA, for more information.

  – Include a third party bean validation provider Java archive (JAR) file in the class path of the embeddable EJB container run time.

Bean validation in RAR modules

WebSphere Application Server validates resource adapter archive (RAR) JavaBeans constraints in compliance with the Java Connector Architecture (JCA) version 1.6 specification.

Resource adapters can specify the validation requirements of configuration properties to the Application Server through annotations in the source code of the resource adapter, constraint specifications in a resource adapter validation descriptor, or a mixture of both. In specifying these constraints, resource adapters can use the built-in bean validation constraints supplied with the Application Server, custom bean validation constraints supplied either by the application developer or a third party, or a mixture of both. Resource adapter developers can apply constraints to the fields and JavaBeans-compliant properties of the following JCA types:

- ResourceAdapter
- ManagedConnectionFactory
- ActivationSpec
- AdministeredObject

At run time, the application server creates instances of bean types declared by the resource adapter. Each instance is validated immediately upon setting its configuration properties, before placing the instance into service.

When validating a RAR bean, the Application Server creates an instance of a validator factory according to the bean validation deployment descriptor discovered by the Application Server. A validator instance is then obtained from the factory and used to validate the bean instance.

If validation fails, the Application Server throws a constraint violation exception and reports all violations to the system log. The effects of the exception for each RAR bean type and problem determination information are documented in the topic, Troubleshooting bean validation in RAR modules.

Note: The Bean Validation specification requires that no more than one `validation.xml` is visible on the class path. This requirement is violated whenever two or more stand-alone RARs provide a validation descriptor. See the section, “RAR bean validation descriptor” in this topic, for more information. When more than one `validation.xml` is visible to the Application Server class loaders, the Application Server or application modules might fail to acquire the default ValidatorFactory and subsequently cannot perform bean validation. For example, the server cannot validate beans of a RAR embedded in an application whenever the embedded RAR lacks a validation configuration, and two or more stand-alone RARs provide configurations. To avoid trouble, install stand-alone RARs that provide a bean validation descriptor as isolated whenever possible.

Built-in constraint annotations

Note: Use built-in constraint annotations to specify the range and mandatory attributes of configuration properties rather than provide custom annotations for the same purpose. The following constraints are useful, but you can use all bean validation built-in constraints. See the topic Bean validation built-in constraints for a complete list of the constraints.

- @Min

  Specifies the minimum value of the configuration property decorated with this annotation. The value must be greater than or equal to the specified minimum.
- @Max

  Specifies the maximum value of the configuration property decorated with this annotation. The value must less than or equal to the specified maximum.
- @Size

Specifies the range of values of the configuration property decorated with this annotation. The value must be greater than or equal to the specified minimum and be less than or equal to the specified maximum.

- @NotNull

  Specifies the value of the configuration property decorated with this annotation must not be null. That is, the property is required.

The following example is a RAR bean class that is decorated with built-in constraint annotations.

The value of the serverName configuration property must not be null, and the value of the instanceCount property must be at least 1 when the Application Server creates and configures an instance of the MyConnector class. Otherwise, a constraint validation exception occurs and, in the case of ResourceAdapter bean, the resource adapter fails to start. See the topic Troubleshooting bean validation in RAR modules for more information.

```
package com.my.company;

@Connector(...)
public class MyConnector implements ResourceAdapter, Serializable
{
 @ConfigProperty(type=java.lang.String.class,defaultValue="WAS")
 private String serverName;

 @NotNull()
 public String getServerName() {return serverName;}

 private Integer instanceCount = 0;

 @Min(value=1)
 public Integer getInstanceCount() {return instanceCount;}
 ...
```

RAR bean validation descriptor

Bean validation constraints can be declared through an XML descriptor supplied by a RAR module. In the simplest case, a RAR validation descriptor consists of the validation configuration declared in the validation.xml file and zero or more XML files that declare RAR bean validation constraints. Files containing constraint declarations are specified in the constraint-mapping elements of the validation configuration (validation.xml).

You must package the validation descriptor in the META-INF directory of a RAR module. Any custom constraint annotation classes that are declared in the validation descriptor must also be packaged in the RAR module.

The following example is a simple RAR validation descriptor that declares constraint metadata like the code shown in the section, “Built-in constraint annotations.”

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
  xmlns=http://jboss.org/xml/ns/javax/validation/configuration
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation=http://jboss.org/xml/ns/javax/validation/configuration validation-configuration-1.0.xsd>
<constraint-mapping>META-INF/constraints.xml</constraint-mapping>
</validation-config>
```

The constraints XML file is also located in the META-INF directory and looks like the following:

```
<constraint-mappings
 xmlns=http://jboss.org/xml/ns/javax/validation/mapping
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd>
<default-package>com.my.company</default-package>
<bean class="MyConnector" ignore-annotations="true">
 <field name="serverName">
  <valid/>
  <!-- @NotNull() -->
  <constraint annotation="javax.validation.constraints.NotNull">
   <message>Value is not null</message>
  </constraint>
 </field>
 <field name="instanceCount">
  <valid/>
```

```
  <!-- @Min(1) -->
  <constraint annotation="javax.validation.constraints.Min">
    <message>Minimum possible value is 1</message>
    <element name="value">1</element>
   </constraint>
  </field>
 </bean>
<constraint-mapping>
```

The packaged RAR module, `MyResourceAdapter.rar`, looks like the following:

```
my/
  company/
    MyConnector.class
. . .
META-INF
  /validation.xml
  /constraints.xml
```

Third-party bean validation

WebSphere Application Server supports using different bean validation implementations. If a resource adapter requires a bean validation implementation different from the implementation that is provided by the product, and the RAR provides the bean validation implementation, you must package the JAR file that contains the bean validation implementation in the RAR module root directory.

The RAR module must also contain a single validation configuration descriptor (`validation.xml`), which can be packaged in the META-INF directory of the RAR module, or in the `META-INF/services` directory of the bean validation JAR file, but not both.

RAR bean validation configuration discovery

When validating RAR beans, the Application Server bootstraps the bean validation configuration, specific to the RAR, according to the bean validation descriptor supplied in the RAR META-INF directory. If the descriptor does not exist, the server bootstraps the configuration using the first validation descriptor discovered in the RAR class loading context, such as that supplied in a third-party bean validation that is packaged in the RAR. Finally, the server uses the default validation configuration provided by the product.

The server then creates a validator factory specific to the discovered bean validation configuration and uses this factory to create validator instances for validating the RAR bean instances. When you deploy a RAR that supplies a bean validation descriptor, you must take additional steps to ensure that the class loader that loads the RAR loads the bean validation descriptor and classes packaged in the RAR.

For an embedded RAR, after you have deployed the application that embeds the RAR, you must set the delegation mode of the application class loader to Parent-Last (Child-First). See the topic Configuring application class loaders for more information.

For a stand-alone RAR, you must install the RAR as an isolated resource provider. See the topic Resource Adapter settings for more information.

Bean validation in JPA

Data validation is a common task that occurs in all layers of an application, including persistence. The Java Persistence API (JPA) 2.0 provides support for the Bean Validation API so that data validation can be done at run time. This topic includes a usage scenario where bean validation is used in the JPA environment of a sample digital image gallery application.

The Bean Validation API provides seamless validation across technologies on Java Enterprise Edition 6 (Java EE 6) and Java Platform, Standard Edition (JSE) environments. In addition to JPA 2.0, these technologies include JavaServer Faces (JSF) 2.0 and Java EE Connector Architecture (JCA) 1.6. You can read more about bean validation in the topic, Bean Validation API.

There are three core concepts of bean validation: constraints, constraint violation handling, and the validator. If you are running applications in an integrated environment like JPA, there is no need to interface directly with the validator.

Validation constraints are annotations or XML code that are added to a class, field, or method of a JavaBeans component. Constraints can be built in or user-defined. They are used to define regular constraint definitions and for composing constraints. The built-in constraints are defined by the bean validation specification and are available with every validation provider. For a list of built-in constraints, see the topic, Bean validation built-in constraints. If you need a constraint different from the built-in constraints, you can build your own user-defined constraint.

Constraints and JPA

The following usage scenario illustrates how a built-in constraint is used in the JPA architecture of a sample digital image gallery application.

In the first code example, a built-in constraint is added to a simple entity of the JPA model called *image*. An image has an ID, image type, file name, and image data. The image type must be specified and the image file name must include a valid JPEG or GIF extension. The code shows the annotated image entity with some built-in bean validation constraints applied.

```
package org.apache.openjpa.example.gallery.model;

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;

@Entity
public class Image {

    private long id;
    private ImageType type;
    private String fileName;
    private byte[] data;

    @Id
    @GeneratedValue
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    @NotNull(message="Image type must be specified.")
    @Enumerated(EnumType.STRING)
    public ImageType getType() {
        return type;
    }

    public void setType(ImageType type) {
        this.type = type;
    }

    @Pattern(regexp = ".*\\.jpg|.*\\.jpeg|.*\\.gif",
        message="Only images of type JPEG or GIF are supported.")
    public String getFileName() {
        return fileName;
    }
```

```
    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public byte[] getData() {
        return data;
    }

    public void setData(byte[] data) {
        this.data = data;
    }
}
```

The Image class uses two built-in constraints, @NotNull and @Pattern. The @NotNull constraint ensures that an ImageType element is specified and the @Pattern constraint uses regular expression pattern matching to ensure that the image file name is suffixed with a supported image format. Each constraint has corresponding validation logic that gets started at run time when the image entity is validated. If either constraint is not met, the JPA provider throws a ConstraintViolationException with the defined message. The JSR-303 specification also makes provisions for the use of a variable within the message attribute. The variable references a keyed message in a resource bundle. The resource bundle supports environment-specific messages and globalization, translation, and multicultural support of messages.

You can create your own custom validator and constraints. In the previous example, the Image entity used the @Pattern constraint to validate the file name of the image. However, it did not check constraints on the actual image data itself. You can use a pattern-based constraint; however, you do not have the flexibility that you would if you created a constraint specifically for checking constraints on the data. In this case you can build a custom method-level constraint annotation. The following is a custom or user-defined constraint called ImageContent.

```
package org.apache.openjpa.example.gallery.constraint;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import javax.validation.Constraint;
import javax.validation.Payload;

import org.apache.openjpa.example.gallery.model.ImageType;

@Documented
@Constraint(validatedBy = ImageContentValidator.class)
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
public @interface ImageContent {
     String message() default "Image data is not a supported format.";
     Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  ImageType[] value() default { ImageType.GIF, ImageType.JPEG };
}
```

Next, you must create the validator class, ImageContentValidator. The logic within this validator gets implemented by the validation provider when the constraint is validated. The validator class is bound to the constraint annotation through the validatedBy attribute on the @Constraint annotation as shown in the following code:

```
package org.apache.openjpa.example.gallery.constraint;
import java.util.Arrays;
import java.util.List;
import javax.validation.ConstraintValidator;
```

```
import javax.validation.ConstraintValidatorContext;
import org.apache.openjpa.example.gallery.model.ImageType;
/**
 * Simple check that file format is of a supported type
 */
public class ImageContentValidator implements ConstraintValidator<ImageContent, byte[]> {
    private List<ImageType> allowedTypes = null;
    /**
     * Configure the constraint validator based on the image
     * types it should support.
     * @param constraint the constraint definition
     */
    public void initialize(ImageContent constraint) {
        allowedTypes = Arrays.asList(constraint.value());
    }
    /**
     *Validate a specified value.
     */
    public boolean isValid(byte[] value, ConstraintValidatorContext context) {
        if (value == null) {
            return false;
        }
        // Verify the GIF header is either GIF87 or GIF89
        if (allowedTypes.contains(ImageType.GIF)) {
            String gifHeader = new String(value, 0, 6);
            if (value.length >= 6 &&
                (gifHeader.equalsIgnoreCase("GIF87a") ||
                 gifHeader.equalsIgnoreCase("GIF89a"))) {
                return true;
            }
        }
        // Verify the JPEG begins with SOI and ends with EOI
        if (allowedTypes.contains(ImageType.JPEG)) {
            if (value.length >= 4 &&
                value[0] == 0xff && value[1] == 0xd8 &&
                value[value.length - 2] == 0xff && value[value.length -1] == 0xd9) {
                return true;
            }
        }
        // Unknown file format
        return false;
    }
}
```

Apply this new constraint to the getData() method on the Image class; for example:

```
@ImageContent
    public byte[] getData() {
        return data;
    }
```

When validation of the data attribute occurs, the isValid() method in the ImageContentValidator is started. This method contains logic for performing simple validation of the format of the binary image data. A potentially overlooked feature in the ImageContentValidator is that it can also validate for a specific image type. By definition, it accepts JPEG or GIF formats, but it can also validate for a specific format. For example, by changing the annotation to the following code example, the validator is instructed to only permit image data with valid JPEG content:

```
@ImageContent(ImageType.JPEG)
    public byte[] getData() {
        return data;
    }
```

Type-level constraints are also a consideration because you might need to validate combinations of attributes on an entity. In the previous examples validation constraints were used on individual attributes. Type-level constraints make it possible to provide collective validation. For example, the constraints

applied to the image entity validate that an image type is set (not null), the extension on the image file name is of a supported type, and the data format is correct for the indicated type. But, for example, it does not collectively validate that a file named `img0.gif` is of type GIF and the format of the data is for a valid GIF file image. For more information about type-level constraints, see the white paper, OpenJPA Bean Validation Primer, and the section "Type-level constraints."

Validation groups

Bean validation uses validation groups to determine what type of validation and when validation occurs.

There are no special interfaces to implement or annotations to apply to create a validation group. A validation group is denoted by a class definition.

Note: When using groups, use simple interfaces. Using a simple interface makes validation groups more usable in multiple environments. Whereas, if a class or entity definition is used as a validation group, it might pollute the object model of another application by bringing in domain classes and logic that do no make sense for the application. By default, if a validation group or multiple groups is not specified on an individual constraint, it is validated using the javax.validation.groups.Default group. Creating a custom group is as simple as creating a new interface definition.

For more information about validation groups, read the white paper, OpenJPA Bean Validation Primer, and the section "Validation groups."

JPA domain model

In addition to the Image entity are Album, Creator and Location persistent types. An Album entity contains a reference to collection of its Image entities. The Creator entity contains a reference to the album entities that the image Creator contributed to and a reference to the Image entities created. This provides full navigational capabilities to and from each of the entities in the domain. An embeddable location, has been added to image to support storing location information with the image.

The Album and Creator entities have standard built-in constraints. The embeddable location is more unique in that it demonstrates the use of the @Valid annotation to validate embedded objects. To embed location into an image, a new field and corresponding persistent properties are added to the Image class; for example:

```
private Location location;

    @Valid
    @Embedded
    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }
```

The @Valid annotation provides chained validation of embeddable objects within a JPA environment. Therefore, when image is validated, any constraints on the location it references are also validated. If @Valid is not specified, the location is not validated. In a JPA environment, chained validation through @Valid is only available for embeddable objects. Referenced entities and collections of entities are validated separately to prevent circular validation.

Bean validation and the JPA environment

The JPA 2.0 specification makes integration with the Bean Validation API simple. In a JSE environment, bean validation is enabled by default when you provide the Bean Validation API and a bean validation

provider on your runtime class path. In a Java EE 6 environment, the application server includes a bean validation provider so there is no need to bundle one with your application. In both environments, you must use a Version 2.0 `persistence.xml` file.

A Version 1.0 `persistence.xml` provides no means to configure bean validation. Requiring a Version 2.0 `persistence.xml` prevents a pure JPA 1.0 application from incurring the validation startup and runtime costs. This is important given that there is no standard means for a 1.0-based application to disable validation. In a Java EE 6 environment, enable validation in an existing 1.0 application by modifying the root element of your `persistence.xml` file. The following example represents the `persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0" >
...
</persistence>
```

Bean validation provides three modes of operation within the JPA environment:

- Auto

  Enables bean validation if a validation provider is available within the class path. Auto is the default.
- Callback

  When callback mode is specified, a bean validation provider must be available for use by the JPA provider. If not, the JPA provider throws an exception upon instantiation of a new JPA entity manager factory.
- None

  Disables bean validation for a particular persistence unit.

Auto mode simplifies deployment, but can lead to problems if validation does not take place because of a configuration problem.

Note: Use either none or callback mode explicitly for consistent behavior.
Also, if none is specified, JPA optimizes at startup and does not attempt to perform unexpected validation. Explicitly disabling validation is especially important in a Java EE 6 environment where the container is mandated to provide a validation provider. Therefore, unless specified, a JPA 2.0 application started in a container has validation enabled. This process adds additional processing during life cycle events.

There are two ways to configure validation modes in JPA 2.0. The simplest way is to add a validation-mode element to the `persistence.xml` with the wanted validation mode as shown in the following example:

```
 <persistence-unit name="auto-validation">
          ...
    <!-- Validation modes: AUTO, CALLBACK, NONE -->
    <validation-mode>AUTO</validation-mode>
    ...
  </persistence-unit>
```

The other way is to configure the validation mode programmatically by specifying the javax.persistence.validation.mode property with value auto, callback, or none when creating a new JPA entity manager factory as shown in the following example:

```
Map<String, String> props = new HashMap<String, String>();
  props.put("javax.persistence.validation.mode", "callback");
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("validation", props);
```

Bean validation within JPA occurs during JPA life cycle event processing. If enabled, validation occurs at the final stage of the PrePersist, PreUpdate, and PreRemove life cycle events. Validation occurs only after

all user-defined life cycle events, since some of those events can modify the entity that is being validated. By default, JPA enables validation for the default validation group for PrePersist and PreUpdate life cycle events. If you must validate other validation groups or enable validation for the PreRemove event, you can specify the validation groups to validate each life cycle event in the `persistence.xml` as shown in the following example:

```
<persistence-unit name="non-default-validation-groups">
  <class>my.Entity</class>
  <validation-mode>CALLBACK</validation-mode>
  <properties>
   <property name="javax.persistence.validation.group.pre-persist"
       value="org.apache.openjpa.example.gallery.constraint.SequencedImageGroup"/>
   <property name="javax.persistence.validation.group.pre-update"
       value="org.apache.openjpa.example.gallery.constraint.SequencedImageGroup"/>
   <property name="javax.persistence.validation.group.pre-remove"
       value="javax.validation.groups.Default"/>
  </property>
 </persistence-unit>
```

The following example shows various stages of the JPA life cycle, including persist, update, and remove:

```
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("BeanValidation");
        EntityManager em = emf.createEntityManager();

        Location loc = new Location();
        loc.setCity("Rochester");
        loc.setState("MN");
        loc.setZipCode("55901");
        loc.setCountry("USA");

        // Create an Image with non-matching type and file extension
        Image img = new Image();
        img.setType(ImageType.JPEG);
        img.setFileName("Winter_01.gif");
        loadImage(img);
        img.setLocation(loc);

        // *** PERSIST ***
        try {
            em.getTransaction().begin();
            // Persist the entity with non-matching extension and type
            em.persist(img);
        } catch (ConstraintViolationException cve) {
            // Transaction was marked for rollback, roll it back and
            // start a new one
            em.getTransaction().rollback();
            em.getTransaction().begin();
            // Fix the file type and re-try the persist.
            img.setType(ImageType.GIF);
            em.persist(img);
            em.getTransaction().commit();
        }

        // *** UPDATE ***
        try {
            em.getTransaction().begin();
            // Modify the file name to a non-matching file name
            // and commit to trigger an update
            img.setFileName("Winter_01.jpg");
            em.getTransaction().commit();
        }  catch (ConstraintViolationException cve) {
            // Handle the exception.  The commit failed so the transaction
            // was already rolled back.
            handleConstraintViolation(cve);
        }
        // The update failure caused img to be detached. It must be merged back
```

```
        // into the persistence context.
        img = em.merge(img);

        // *** REMOVE ***
        em.getTransaction().begin();
        try {
            // Remove the type and commit to trigger removal
            img.setType(ImageType.GIF);
            em.remove(img);
        }  catch (ConstraintViolationException cve) {
            // Rollback the active transaction and handle the exception
            em.getTransaction().rollback();
            handleConstraintViolation(cve);
        }
        em.close();
        emf.close();
```

Exceptions

Validation errors can occur in any part of JPA life cycle.

If one or more constraints fail to validate during a life cycle event, a ConstraintViolationException is thrown by the JPA provider. The ConstraintViolationException thrown by the JPA provider includes a set of ConstraintViolations that occurred. Individual constraint violations contain information regarding the constraint, including: a message, the root bean or JPA entity, the leaf bean which is useful when validating JPA embeddable objects, the attribute which failed to validate, and the value that caused the failure. The following is a sample exception handling routine:

```
private void handleConstraintViolation(ConstraintViolationException cve) {
      Set<ConstraintViolation<?>> cvs = cve.getConstraintViolations();
   for (ConstraintViolation<?> cv : cvs) {
     System.out.println("------------------------------------------------");
           System.out.println("Violation: " + cv.getMessage());
           System.out.println("Entity: " + cv.getRootBeanClass().getSimpleName());
           // The violation occurred on a leaf bean (embeddable)
           if (cv.getLeafBean() != null && cv.getRootBean() != cv.getLeafBean()) {
              System.out.println("Embeddable: " +
cv.getLeafBean().getClass().getSimpleName());
           }
           System.out.println("Attribute: " + cv.getPropertyPath());
           System.out.println("Invalid value: " + cv.getInvalidValue());
      }
    }
```

Constraint violation processing is typically simple when using attribute-level constraints. If you are using a type-level validator with type-level constraints, it can be more difficult to determine which attribute or combination of attributes failed to validate. Also, the entire object is returned as the invalid value instead of an individual attribute. In cases where specific failure information is required, use of an attribute-level constraint or a custom constraint violation might be provided as described in the Bean Validation specification.

Sample

The JPA model and image gallery application usage scenario provided in this topic can be implemented through a sample that is provided in the white paper, *OpenJPA Bean Validation primer*.

Chapter 5. Developing Client applications

This page provides a starting point for finding information about application clients and client applications. Application clients provide a framework on which application code runs, so that your client applications can access information on the application server.

For example, an insurance company can use application clients to help offload work on the server and to perform specific tasks. Suppose an insurance agent wants to access and compile daily reports. The reports are based on insurance rates that are located on the server. The agent can use application clients to access the application server where the insurance rates are located. More introduction...

Developing client applications

A *client application* performs business logic and makes use of the framework provided by an underlying *client*. Developing the code for a client application depends on the objects and functions you want to exploit, and the programming model that you want to use.

Before you begin

Install the software development resources needed to develop client applications for use with WebSphere Application Server. During code development, you do not need access to the WebSphere Application Server. However, to assemble some types of client applications you need to install files for the client that provides the framework for the client application. Instead of installing WebSphere Application Server you can install the Application Client feature, which provides the same resources and clients to aid development of client applications.

About this task

To use a client application to access a remote object on an application server, develop your client application code as described in the following steps and the related topics. These topics only describe the client-specific considerations; they do not describe general client programming models, which you should already be familiar with. Samples for different types of client applications are provided with the Application Client.

Procedure

1. Choose the type of client that you want to use as a framework for your client application. Decision factors for choosing a client include whether you want to run a client application on Java EE or J2SE; whether you want ease of use with a small installation footprint or full-function with medium-large footprint; and whether you need licence to copy or redistribute the client. For more information about choosing the type of client, see “Choosing a type of client” on page 46.
2. Develop the client application code. The following substeps are a high-level general procedure. Information specific to a type of client is given in the related tasks.
   a. Create an instance of the object that you want to access on the remote server. You can use full Java Naming and Directory Interface (JNDI) support to get a suitable reference to administered objects from the server's JNDI namespace. Alternatively, you can get suitable references to objects programmatically without using JNDI.

      Using the javax.naming.InitialContext class, the client application program uses the lookup operation to access the Java Naming and Directory Interface (JNDI) namespace. The InitialContext class provides the lookup method to locate resources.

      You can compare the use of JNDI and programmatic techniques by looking at the samples provided for the Java EE client and Java thin client in an Application Client installation (for example, in `C:\wac70\samples\src\`):

- Java EE client use of JNDI for BasicCalculatorHome: `TechnologySamplesJ2EEClient\BasicCalculator\com\ibm\websphere\samples\technologysamples\basiccalcclient\BasicCalculatorClientJ2EE.java`
- Java thin client programmatic retrieval of BasicCalculatorHome: `TechnologySamplesThinClient\BasicCalculator\com\ibm\websphere\samples\technologysamples\basiccalcthinclient\BasicCalculatorClientThin.java`

b. Create a connection to the server. If the server runs with security enabled, you can configure secure connections.

c. Work with the objects to perform your business requirements. For example, send and receive JMS messages, update database entries, handle error conditions, and close resources used.

3. Compile or assemble the client application. This creates the JAR or EAR file that you can deploy to make the client application available for use.

   To compile your client application, include the JAR files needed in the CLASSPATH setting for the **javac** command; for example, any extra JAR files for the client application's own classes, JAR files for IBM® Thin clients used, and JAR files for JDBC provider classes.

   Attention: IBM-provided clients are not packaged with JDBC provider classes. For example, the WebSphere Application Server Version 7.0 Java Thin application client is not packaged with Apache Derby 10.2 classes. If your client application needs to use a database class (such as through the JNDI lookup of a datasource), you must obtain the class files from the database provider and make them available when compiling and running the client application.

What to do next

After you develop a client application, deploy it into the environment you want it to run.

Choosing a type of client

A *client* provides the framework for client applications that run separately from your application server.

About this task

Decision factors for choosing a client include whether you want to run a client application on Java Platform, Enterprise Edition (Java EE) or Java Platform, Standard Edition (J2SE) ; whether you want ease of use with a small installation footprint or full-function with medium to large footprint; and whether you need licence to copy or redistribute the client.

A usual first decision to make is do you want a client application that runs on Java EE or J2SE?. This leads you to choose from the main types of clients, as described in this topic. Otherwise, if you want to run an ActiveX program, or a Java applet, to interact with enterprise beans on WebSphere Application Server, your decision is only for one of those types of client.

Procedure

- J2SE

  If you want to run a lightweight client application, without the resource and processing cost of the Java EE platform for WebSphere Application Server on the client machine, then choose either the Java thin client or the stand-alone thin clients to run on J2SE.

  – If you want a client with a small installation footprint, that you can embed into your application, and that runs under an IBM, Sun, or HP-UX JRE, choose the stand-alone clients. Each client is an embeddable single jar with small footprint; for example, the Thin Client for JMS with WebSphere Application Server, `com.ibm.ws.sib.client.thin.jms_7.0.0.jar` needs about 2 MB of disk space. For notable restrictions of stand-alone clients, see the client comparison table in Client applications.

  – If you want a full-function client with medium to large footprint, that runs under the IBM JRE supplied, choose the Java thin client.

- If you run your client application to use the installed files of Application Client for WebSphere Application Server, you need about 400 MB of disk space (as part of the Application Client installation). Choose this option if you intend to copy and redistribute the Java thin client, within your licensing agreement.
- If you run your client application to use the installed files of the WebSphere Application Server, you need about 1 GB of disk space (as part of the Application Server installation). Choose this option if you do not mind the larger footprint, and you want maintenance support for the Java thin client.

However, the thin clients running on J2SE do not support a Java EE container that provides easy access to system services for object resolution, security, Reliability Availability and Servicability (RAS), and other services. Also, thin clients running on J2SE do not initialize any of the services that the client application might require.

- Java EE

  If you want to run a Java client application that makes full use of the Java EE platform features of WebSphere Application Server, then choose the Java EE client.

  – If you run your client application to use the installed files of Application Client for WebSphere Application Server, you need about 400 MB of disk space (as part of the Application Client installation). Choose this option if you intend to copy and redistribute the Java EE client, within your licensing agreement.
  – If you run your client application to use the installed files of the WebSphere Application Server, you need about 1 GB of disk space (as part of the Application Server installation). Choose this option if you do not mind the larger footprint, and you want maintenance support for the Java EE client.

  The Java EE client provides a container that client applications can use to access system services. The Java EE client also initializes the runtime environment for client applications.
- ActiveX to Enterprise JavaBeans (EJB) Bridge

  If you want ActiveX programs to access enterprise beans on WebSphere Application Server, choose this client.
- Applet client

  If you want a browser-based Java client application program that provides a richer and more robust environment than the one offered by the Applet > Servlet > enterprise bean model, choose this client.

What to do next

Develop your client application to use the type of client that you have chosen.

Developing stand-alone thin client applications

Develop the application code, then assemble the code into a client application that you can deploy on a client machine.

Procedure

- Get server objects and resources.

  A stand-alone client application can get suitable server objects and resources (like connection factories, JMS queues, and data sources) programmatically without using JNDI. Alternatively, a client application can use full JNDI support provided by the Thin Client for EJB.
- Compile stand-alone thin client applications. To compile your client application, include the JAR files needed in the CLASSPATH setting for the **javac** command; for example, any extra JAR files for the client application's own classes, JAR files for IBM Thin clients used, and JAR files for JDBC provider classes.

  For the stand-alone thin clients, the following JAR files are provided in the `/runtimes/` directory of either an Application Client installation or Application Server installation:

Table 2. JAR files for stand-alone thin clients. The product provides JAR files for stand-alone thin clients descriptions.

JAR file	Description
com.ibm.jaxws.thinclient_8.5.0.jar	IBM Thin Client for Java API for XML-based Web Services (JAX-WS). This file enables a Java SE client application to use the JAX-WS programming model to invoke web services that are hosted by the application server. You must use the endorsed APIs JAR file when starting Java because the Thin Client for JAX-WS requires APIs that are more current than what is available in JDKs to support JAX-WS 2.2 and JAXB 2.2 implementations.
com.ibm.ws.ejb.thinclient_8.5.0.jar	Thin Client for Enterprise Java Beans (EJB) . This file enables a Java SE client application to access remote Enterprise Java Beans on a server through Java Naming and Directory Interface (JNDI) look up. If this file is running with a non-IBM product JRE on a non-IBM product platform, the IBM ORB implementation library, com.ibm.ws.orb_8.5.0.jar, is also needed.
com.ibm.ws.jpa.thinclient_8.5.0.jar	IBM Thin Client for Java Persistence API (JPA). This file allows a Java SE client application to use the Java Persistence API (JPA) to store and retrieve persistent data without the use of an application server.
com.ibm.ws.messagingClient.jar	With the com.ibm.ws.ejb.thinclient_8.5.0.jar file, this file enables a Java SE client application to use WebSphere MQ messaging provider JMS resources from the WebSphere Application Server JNDI namespace. WebSphere MQ client jar files are also needed, and must be obtained from the WebSphere MQ product.
com.ibm.ws.orb_8.5.0.jar	The IBM ORB implementation library. This file is needed if the IBM Thin Client for EJB is running with a non-IBM product JRE on a non-IBM product platform.
com.ibm.ws.sib.client.thin.jms_8.5.0.jar	IBM Thin Client for Java Messaging Service (JMS). This file enables a Java SE client application to use JMS resources of the default messaging provider. For languages other than US English, you also need the additional language files from sibc.nls.zip, which provides language-specific resource bundles.
com.ibm.ws.sib.client_ExpeditorDRE_8.5.0.jar	The JMS Client packaged for Lotus® Expeditor.
com.ibm.ws.webservices.thinclient_8.5.0.jar	IBM Thin Client for Java API for XML-based RPC (JAX-RPC). This file enable a Java SE client application to use the JAX-RPC programming model to invoke web services that are hosted by the application server.

If you are running two or more of these stand-alone thin clients together, you must obtain all the clients that you are using from the same installation of Application Client for WebSphere Application Server, the same installation of the WebSphere Application Server product, or the same service refresh.

What to do next

After developing and compiling a stand-alone thin client application, you can deploy and run the client application.

Using JMS resources

If you are using JMS resources with the Thin Client for JMS with WebSphere Application Server, you can choose either to obtain these resources programmatically or using the Java Naming and Directory Interface (JNDI). Stand-alone Java SE JMS thin client applications that connect to an external WebSphere MQ queue manager can get administratively-created WebSphere MQ messaging provider JMS resources from the WebSphere Application Server Java Naming and Directory Interface (JNDI) namespace.

About this task

If you are using the Thin Client for JMS with WebSphere Application Server, you can obtain suitable JMS connection factories and references to JMS queues or topics programmatically without using JNDI. Alternatively, full JNDI support might be obtained from the Thin Client for EJB with WebSphere Application Server. For further information, refer to the Using JMS resources with the Thin Client for JMS with WebSphere Application Server topic.

If you are using a stand-alone Java SE JMS thin client application that connects to an external WebSphere MQ queue manager and want to obtain administratively-created WebSphere MQ messaging provider JMS resources from the WebSphere Application Server JNDI namespace, refer to the Obtaining WebSphere MQ JMS resources in the thin client environment topic.

Developing a Java EE client application

This topic provides the steps that are required to develop code for a Java Platform, Enterprise Edition (Java EE) client application.

About this task

Procedure

1. Write the client application program. Write the Java EE client application on any development machine. At this stage, you do not require access to the WebSphere Application Server.

   Rules: If you are writing a client application program that will run on z/OS®, the following rules apply:
   - Client programs may start their own transactions but cannot join in or start transactions in the WebSphere Application Server for z/OS run-time.
   - Client application code must contain a main method.
   - All input and output files for the client application must be in ASCII, because the client run-time runs in an ASCII JVM.

2. Assemble the client application JAR file using an assembly tool.

   The JNDI namespace knows what to return on a lookup because of the information assembled by the assembly tool.

   Assemble the client application on any development machine with the assembly tool installed.

   When you assemble your client application, provide the required information to initialize the runtime environment for your client application. For information about how to provide the required information, see the documentation for the assembly tool.

   When you configure resources for use by your client application, consider the following items:
   - Resource environment references are different than resource references. Resource environment references enables your client application to use a logical name to look up a resource bound into the server JNDI namespace. A resource reference enables your application to use a logical name to look up a local Java EE resource. The Java EE specification does not specify a particular implementation of a resource. The following table contains supported resource types and identifies the resources to which the WebSphere Application Server provides a client implementation.

Table 3. Supported resource types and resource identifiers. Supported resource types

Resource Type	Client Configuration Notes®	Client implementation provided by WebSphere Application Server
javax.sql.DataSource	Supports specification of any data source implementation class	No
java.net.URL	Supports specification of custom protocol handlers	Provided by Java Runtime Environment files
javax.mail.Session	Supports custom protocol configuration	Yes - POP3/POP3S, SMTP/SMTPS, IMAP/IMAPS
javax.jms.QueueConnectionFactory, javax.jms.TopicConnectionFactory, javax.jms.Queue, javax.jms.Topic	Supports configuration of WebSphere embedded messaging, IBM MQ Series and other JMS providers	Yes - WebSphere embedded messaging

3. Assemble the Enterprise Archive (EAR) file.

   The application is contained in an enterprise archive (EAR file). The EAR file is composed of:
   - Enterprise bean, application client, and user-defined modules or JAR files
   - Web applications or WAR files
   - Metadata describing the applications or application XML files

What to do next

After developing the Java EE client application code, deploy the application onto the client machines where the client application is to run.

Java EE client application class loading

When you run your Java Platform, Enterprise Edition (Java EE) application client, a hierarchy of class loaders is created to load classes used by your application.

The following list describes the hierarchy of class loaders:

- The Application Client for WebSphere Application Server (Application Client) run time sets this value to the `WAS_LOGGING` environment variable.
- The extensions class loader is a child to the bootstrap class loader. This class loader contains JAR files in the `java/jre/lib/ext` directory or those JAR files defined by the `-Djava.ext.dirs` parameter on the Java command. The Application Client client run time does not set `-Djava.ext.dirs` parameters. So it uses the JAR files in the `java/jre/lib/ext` directory.
- The system class loader contains JAR files and classes that are defined by the `-classpath` parameter on the Java command. The Application Client run time sets this parameter to the `WAS_CLASSPATH` environment variable.
- The WebSphere class loader loads the Application Client run time and any classes placed in the Application Client user directories. The directories used by this class loader are defined by the `WAS_EXT_DIRS` environment variable. The `WAS_BOOTCLASSPATH`, `WAS_CLASSPATH`, and the `WAS_EXT_DIRS` environment variables are set in the *app_server_root*`/bin/setupCmdLine` script for WebSphere Application Server installations, or in the *app_server_root*`/bin/setupClient` script for client installations.

As the Java EE application client run time initializes, additional class loaders are created as children of the WebSphere class loader. If your client application uses resources such as Java DataBase Connectivity (JDBC) API, Java Message Service (JMS) API, or Uniform Resource Locator (URL), a different class loader is created to load each of those resources. Finally, the Application Client run time sets the WebSphere class loader to load classes within the EAR file by processing the client JAR manifest repeatedly. The system class path, defined by the CLASSPATH environment variable is never used and is not part of the hierarchy of class loaders.

To package your client application correctly, you must understand which class loader loads your classes. When the Java code loads a class, the class loader used to load that class is assigned to it. Any classes subsequently loaded by that class will use that class loader or any of its parents, but it will not use children class loaders.

In some cases the Application Client run time can detect when your client application class is loaded by a different class loader from the one created for it by the Application Client run time. When this detection occurs, you see the following message:

```
WSCL0205W: The incorrect class loader was used to load [0]
```

This message occurs when your client application class is loaded by one of the parent class loaders in the hierarchy. This situation is typically caused by having the same classes in the EAR file and on the hard drive. If one of the parent class loaders locates a class, that class loader loads it before the Application Client run time class loader. In some cases, your client application still functions correctly. In most cases, however, you receive “class not found” exceptions.

Configuring the classpath fields

When packaging your Java EE client application, you must configure various class path fields. Ideally, you should package everything required by your application into your EAR file. This is the easiest way to distribute your Java EE client application to your clients. However, you should not package such resources as JDBC APIs, JMS APIs, or URLs. In the case of these resources, use class path references to access those classes on the hard drive. You might also have other classes installed on your client machines that

you do not need to redistribute. In this case, you also want to use classpath references to access the classes on the hard drive, as described later in this topic.

Referencing classes within the EAR file

WebSphere product Java EE applications do not use the system class path. Use the MANIFEST Class path entry to refer to other JAR files within the EAR file. Configure these values using an assembly tool. For example, if your client application needs to access the path of the EJB JAR file, add the deployed enterprise bean module name to your application client class path. The format of the Class path field for each of the different modules (Application Client, EJB, Web) is the same:

- The values must refer to JAR and class files that are contained within the EAR file.
- The values must be relative to the root of the EAR file.
- The values cannot refer to absolute paths in the file systems.
- Multiple values must be separated by spaces, not colons or semicolons.

Attention: This is the Java method for allowing applications to function platform independent.

Typically, you add modules (JAR files) to the root of the EAR file. In this case, you only need to specify the name of the module (JAR file) in the Class path field. If you choose to add a module with a path, you need to specify the path relative to the root of the EAR file.

For referencing class files, you must specify the directory relative to the root of the EAR file. With an assembly tool, you can add individual class files to the EAR file. It is recommended that these additional class files are packaged in a JAR file. Add this JAR file to the module Class path fields. If you add class files to the root of the EAR file, add `./` to the module Class path fields.

Consider the following example directory structure in which the file myapp.ear contains an application client JAR file named `myclient.jar` and a `mybeans.jar` EJB module. Additional classes reside in `class1.jar` and `utility/class2.zip` files. A class file named `xyz.class` is not packaged in a JAR file but is in the root of the EAR file. Specify `./ mybeans.jar utility/class2.zip class1.jar` as the value of the Classpath property. The search order is: `myapp.ear/myclient.jar myapp.ear/xyz.class myapp.ear/mybeans.jar myapp.ear/utility/class2.zip myapp.ear/class1.jar`

Referencing classes that are not in the EAR file

Use the launchClient `-CCclasspath` parameter. This parameter is specified at run time and takes platform-specific class path values, which means multiple values are separated by semi-colons or colons. The client and the server are similar in this respect.

Resource class paths

When you configure resources used by your client application using the Application Client Resource Configuration Tool (ACRCT), or the z/OS ACRCT scripting tool, you can specify class paths that are required by the resource. For example, if your application is using a JDBC to a DB2® database, add `db2java.zip` to the class path field of the database provider. These class path values are platform-specific and require semi-colons or colons to separate multiple values.

On WebSphere Application Server for i5/OS®, if you use the IBM Developer Kit for Java JDBC provider to access DB2/400, you do not have to add the `db2_classes.jar` file to the class path. However, if you use the IBM Toolbox for Java JDBC provider, specify the location of the `jt400.jar` file.

Using the launchClient API

If you use the **`launchClient`** command, the WebSphere class loader hierarchy is created for you. However, if you use the launchClient API, you must perform this setup yourself. Copy the `launchClient` shell command in defining the Java system properties.

Assembling Java EE client applications

Application client projects contain programs that run on networked client systems. An application client project is deployed as a Java archive (JAR) file.

About this task

Assemble a client module to contain client application code. Group enterprise beans, web components, and resource adapter code in separate modules.

Use an assembly tool to assemble an application client module in any of the following ways:
- Import an existing application client JAR file.
- Create a new application client module.

Procedure

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules. Ensure that Java EE capability is enabled.
3. Migrate application client JAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an assembly tool. To migrate files, import your application client JAR files to the assembly tool.
4. Create a new client application.
5. Verify the contents of the new client application in either of the following ways:
   - In the Project Explorer view, expand **Application Client Projects** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the application client module in a Navigator view.

What to do next

After you finish assembling all of your application's modules, you are ready to deploy your application.

Developing a Java thin client application

Develop the application code, then assemble the code into a client application that you can deploy on a client machine.

About this task

To develop a Java thin client application, you developing the application code, generate the client bindings needed for the enterprise bean and CORBA objects, and package these pieces together to install on the client machine.

With the Java thin client, the client application must code explicitly the fully-qualified location for each resource that it uses. For example, a Java thin client application that looks up an enterprise bean Home contains the following code:

```
java.lang.Object ejbHome =
  initialContext.lookup("the/fully/qualified/path/to/actual/home/in/namespace/MyEJBHome");
MyEJBHome = (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome,  MyEJBHome.class);
```

The Java thin client application must know the fully-qualified physical location of the enterprise bean Home in the namespace. If this location changes, the thin client application must also change the value placed on the lookup() statement.

To compile a Java thin client application, include the client jars file needed by the application in the CLASSPATH setting for the javac command.

Developing ActiveX client application code

This topic provides an outline for developing an ActiveX Windows program, such as Visual Basic, VBScript, and Active Server Pages, to use the WebSphere ActiveX to EJB bridge to access enterprise beans.

Before you begin

Important: This topic assumes that you are familiar with ActiveX programming and developing on the Windows platform. For information about the programming concepts of ActiveX application clients and the ActiveX to EJB bridge, refer to the ActiveX to Enterprise JavaBeans™ (EJB) Bridge topic, and related topics.

Consider the information given in ActiveX to EJB bridge as good programming guidelines.

About this task

To use the ActiveX to EJB bridge to access a Java class, develop your ActiveX program to complete the following steps:

Procedure

1. Create an instance of the XJB.JClassFactory object.
2. Create Java virtual machine (JVM) code within the ActiveX program process, by calling the XJBInit() method of the XJB.JClassFactory object. After the ActiveX program has created an XJB.JClassFactory object and called the XJBInit() method, the JVM code is initialized and ready for use.
3. Create a proxy object for the Java class, by using the XJB.JClassFactory FindClass() and NewInstance() methods. The ActiveX program can use the proxy object to access the Java class, object fields, and methods.
4. Call methods on the Java class, using the Java method invocation syntax, and access Java fields as required.
5. Use the helper functions to do the conversion in cases where automatic conversion is not possible. You can convert between the following data types:
   - Java Byte and Visual Basic Byte
   - Visual Basic Currency types and Java 64-bit
6. Implement methods to handle any errors returned from the Java class. In Visual Basic or VBScript, use the Err.Number and Err.Description fields to determine the actual Java error.

Example

- Viewing a System.out message
- ActiveX client application using helper methods for data type conversion

Viewing a System.out message: The ActiveX to Enterprise JavaBeans (EJB) bridge does not have a console available to view Java System.out messages. To view these messages when running a stand-alone client program (such as Visual Basic), redirect the output to a file.

The following example illustrates how to redirect output to a file:

```
launchClientXJB.bat MyProgram.exe > output.txt
```

- To view the System.out messages when running a Service program such as Active Server Pages, override the Java System.out OutputStream object to FileOutputStream. For example, in VBScript:

```
'Redirect system.out to a file
' Assume that oXJB is an initialized XJB.JClassFactory object
 Dim clsSystem
 Dim oOS
 Dim oPS
 Dim oArgs
```

```
' Get the System class
  Set clsSystem = oXJB.FindClass("java.lang.System")

' Create a FileOutputStream object
' Create a PrintStream object and assign to it our FileOutputStream
  Set oArgs = oXJB.GetArgsContainer  oArgs.AddObject "java.io.OutputStream", oOS
  Set oPS = oXJB.NewInstance(oXJB.FindClass("java.io.PrintStream"), oArgs)

' Set our System OutputStream to our file
  clsSystem.setOut oPS
```

ActiveX client application using helper methods for data type conversion. Generally, data type conversion between ActiveX (Visual Basic and VBScript) and Java methods occurs automatically, as described in ActiveX to EJB bridge, converting data types. However, the byte helper function and currency helper function are provided for cases where automatic conversion is not possible.

- Byte helper function

  Because the Java Byte data type is signed (-127 through 128) and the Visual Basic Byte data type is unsigned (0 through 255), convert unsigned Bytes to a Visual Basic Integers, which look like the Java signed byte. To make this conversion, you can use the following helper function:

```
Private Function GetIntFromJavaByte(Byte jByte) as Integer
      GetIntFromJavaByte = (CInt(jByte) + 128) Mod 256 - 128
End Function
```

- Currency helper function

  Visual Basic 6.0 cannot properly handle 64-bit integers like Java methods can (as the Long data type). Therefore, Visual Basic uses the Currency type, which is intrinsically a 64-bit data type. The only side effect of using the Currency type (the Variant type VT_CY) is that a decimal point is inserted into the type. To extract and manipulate the 64-bit Long value in Visual Basic, use code like the following example. For more details on this technique for converting Currency data types, see Q189862, “HOWTO: Do 64-bit Arithmetic in VBA”, on the Microsoft Knowledge Base.

```
' Currency Helper Types
Private Type MungeCurr
    Value As Currency
End Type
Private Type Munge2Long
    LoValue As Long
    HiValue As Long
End Type

' Currency Helper Functions
Private Function CurrToText(ByVal Value As Currency) As String
    Dim Temp As String, L As Long
    Temp = Format$(Value, "#.0000")
    L = Len(Temp)
    Temp = Left$(Temp, L - 5) & Right$(Temp, 4)
    Do While Len(Temp) > 1 And Left$(Temp, 1) = "0"
        Temp = Mid$(Temp, 2)
    Loop
    Do While Len(Temp) > 2 And Left$(Temp, 2) = "-0"
        Temp = "-" & Mid$(Temp, 3)
    Loop
    CurrToText = Temp
End Function

Private Function TextToCurr(ByVal Value As String) As Currency
    Dim L As Long, Negative As Boolean
    Value = Trim$(Value)
    If Left$(Value, 1) = "-" Then
        Negative = True
        Value = Mid$(Value, 2)
    End If
```

```
    L = Len(Value)
    If L < 4 Then
        TextToCurr = CCur(IIf(Negative, "-0.", "0.") & _
                          Right$("0000" & Value, 4))
    Else
        TextToCurr = CCur(IIf(Negative, "-", "") & _
                          Left$(Value, L - 4) & "." & Right$(Value, 4))
    End If
End Function

' Java Long as Currency Usage Example
Dim LC As MungeCurr
Dim L2 As Munge2Long

' Assign a Currency Value (really a Java Long)
' to the MungeCurr type variable
LC.Value = cyTestIn

' Coerce the value to the Munge2Long type variable
LSet L2 = LC

' Perform some operation on the value, now that we
' have it available in two 32-bit chunks
L2.LoValue = L2.LoValue + 1

' Coerce the Munge value back into a currency value
LSet LC = L2
cyTestIn = LC.Value
```

What to do next

After you develop the ActiveX client application code, deploy and run the ActiveX application.

Example: Using an ActiveX client application to access a Java class or object

This reference topic provides an example of using Java proxy objects with the ActiveX to Enterprise JavaBeans (EJB) bridge.

To use Java proxy objects with the ActiveX to Enterprise JavaBeans (EJB) bridge:

- After an ActiveX client program (Visual Basic, VBScript, or Active Server Pages (ASP)) has initialized the XJB.JClassFactory object and thereby, the Java virtual machine (JVM), the client program can access Java classes and initialize Java objects. To complete this action, the client program uses the XJB.JClassFactory FindClass() and NewInstance() methods.
- In Java programming, two ways exists to access Java classes: direct invocation through the Java compiler and through the Java Reflection interface. Because the ActiveX to Java bridge needs no compilation and is a complete run-time interface to the Java code, the bridge depends on the latter Reflection interface to access its classes, objects, methods and fields. The XJB.JClassFactory FindClass() and NewInstance() methods behave very similarly to the Java Class.forName() and the Method.invoke() and Field.invoke() methods.
- XJB.JClassFactory.FindClass() takes the fully qualified class name as its only parameter and returns a Proxy Object (JClassProxy). You can use the returned Proxy object like a normal Java Class object and call static methods and access static fields. You can also create a Class Instance (or object), as described later in this section. For example, the following Visual Basic code extract returns a Proxy object for the java.lang.Integer Java class:

  ```
  ...
  Dim clsMyString as Object
  Set clsMyString = oXJB.FindClass("java.lang.Integer")
  ```

- After the proxy is created, you can access its static information directly. For example, you can use the following code extract to convert a decimal integer to its hexadecimal representation:

```
...
Dim strHexValue as String
strHexValue = clsMyString.toHexString(CLng(255))
```

- The equivalent Java syntax is: `static String toHexString(int i)`. Because ints units in Java programming are really 32-bit (which translates to Long in Visual Basic), the CLng() function converts the value from the default int to a long. Also, even though the toHexString() function returns a java.lang.String, the code extract does not return an Object proxy. Instead, the returned java.lang.String is automatically converted to a native Visual Basic string.

  To create an object from a class, you use the JClassFactory.NewInstance() method. This method creates an Object instance and takes whatever parameters your class constructor needs. Once the object is created, you have access to all of its public instance methods and fields. For example, you can use the following Visual Basic code extract to create an instance of the java.lang.Integer string:

```
...
Dim oMyInteger as Object
set oMyInteger = oXJB.NewInstance(CLng(255))

Dim strMyInteger as String
strMyInteger = oMyInteger.toString
```

Example: ActiveX client application calling Java methods

In the ActiveX to Enterprise Java Beans (EJB) bridge, methods are called using the native language method invocation syntax.

The following differences between Java invocation and ActiveX Automation invocation exist:

- Unlike Java methods, ActiveX does not support method (and constructor) polymorphism; that is, you cannot have two methods in the same class with the same name.
- Java methods are case-sensitive, but ActiveX Automation is not case-sensitive.
- To compensate for Java polymorphic behavior, give the exact parameter types to the method call. The parameter types determine the correct method to invoke. For a listing of correct types to use, see ActiveX to EJB bridge, converting data types.
- For example, the following Visual Basic code fails if the CLng() method was not present or the toHexString syntax was incorrectly typed as ToHexString:

```
...
Dim strHexValue as String
strHexValue = clsMyString.toHexString(CLng(255))
```

- Sometimes it is difficult to force some development environments to leave the case of your method calls unchanged. For example, in Visual Basic if you want to call a method close() (lowercase), the Visual Basic code capitalizes it “Close()”. In Visual Basic, the only way to effectively work around this behavior is to use the CallByName() method. For example:

```
o.Close(123)                                'Incorrect...
CallByName(o, "close", vbMethod, 123)       'Correct...
```

  or in VBScript, use the Eval function:

```
o.Close(123)                                 'Incorrect...
Eval("o.Close(123)")                         'Correct...
```

- The return value of a function is always converted dynamically to the correct type. However, you must take care to use the set keyword in Visual Basic. If you expect a non-primitive data type to return, you must use set. (If you expect a primitive data type to return, you do not need to use set.) See the following example for more explanation:

```
Set oMyObject = o.getObject
iMyInt = o.getInt
```

- In some cases, you might not know the type of object returning from a method call, because wrapper classes are converted automatically to primitives (for example, java.lang.Integer returns an ActiveX Automation Long). In such cases, you might need to use your language built-in exception handling techniques to try to coerce the returned type (for example, On Error and Err.Number in Visual Basic).
- Methods with character arguments

Because ActiveX Automation does not natively support character types supported by Java methods, the ActiveX to EJB bridge uses strings (byte or VT_I1 do not work because characters have multiple bytes in Java code). If you try to call a method that takes a char or java.lang.Character type you must use the JMethodArgs argument container to pass character values to methods or constructors. For more information about how this argument container is used, see Methods with “Object” Type as Argument and Abstract Arguments.

- Methods with “Object” Type as Argument and Abstract Arguments

  Because of the polymorphic nature of Java programming, the ActiveX to Java bridge uses direct argument type mapping to find a method. This method works well in most cases, but sometimes methods are declared with a Parent or Abstract class as an argument type (for example, java.lang.Object). You need the ability to send an object of arbitrary type to a method. To acquire this ability, you must use the XJB.JMethodArgs object to coerce your parameters to match the parameters on your method. You can get a JMethodArgs instance by using the JClassFactory.GetArgsContainer() method.

  The JMethodArgs object is a container for method parameters or arguments. This container enables you to add parameters to it one-by-one and then you can send the JMethodArgs object to your method call. The JClassProxy and JObjectProxy objects recognize the JMethodArgs object and attempt to find the correct method and let the Java language coerce your parameters appropriately.

  For example, to add an element to a Hashtable object the method syntax is `Object put(Object key, Object value)`. In Visual Basic, the method usage looks like the following example code:

```
Dim oMyHashtable as Object
Set oMyHashtable = _
    oXJB.NewInstance(oXJB.FindClass("java.utility.Hashtable"))

' This line will not work. The ActiveX to EJB bridge cannot find a method
' called "put" that has a short and String as a parameter:
oMyHashtable.put 100, "Dogs"
oMyHashtable.put 200, "Cats"

' You must use a XJB.JMethodArgs object instead:
Dim oMyHashtableArgs as Object
Set oMyHashtableArgs = oXJB.GetArgsContainer
oMyHashtableArgs.AddObject("java.lang.Object", 100)
oMyHashtableArgs.AddObject("java.lang.Object", "Dogs")

oMyHashtable.put oMyHashTableArgs
' Reuse the same JMethodArgs object by clearing it.
oMyHashtableArgs.Clear
oMyHashtableArgs.AddObject("java.lang.Object", 200)
oMyHashtableArgs.AddObject("java.lang.Object", "Cats")

oMyHashtable.put oMyHashTableArgs
```

ActiveX client programming best practices

The best way to access Java components is to use the Java language. It is recommended that you do as much programming as possible in the Java language and use a small simple interface between your COM Automation container (for example, Visual Basic) and the Java code. This interface avoids any overhead and performance problems that can occur when moving across the interface.

best-practices: The following topics are covered:

- Visual Basic guidelines
- CScript and Windows Scripting Host
- Active Server Pages guidelines
- J2EE guidelines

Visual Basic guidelines

The following guidelines are intended to help optimize your use of the ActiveX to EJB bridge with Visual Basic:

- Launch the Visual Basic replication through the `launchClientXJB.bat` file. If you want to run your Visual Basic application through the Visual Basic debugger, run the Visual Basic integrated development environment (IDE) within the ActiveX to EJB bridge environment. After you create your Visual Basic project, you can launch it from a command line; for example, `launchClientXJB MyApplication.vbp`. You can also launch the Visual Basic application alone in the ActiveX to EJB environment, by changing the Visual Basic shortcut on the Windows Start menu so that the `launchClientXJB.bat` file precedes the call to the `VB6.EXE` file.
- Exit the Visual Basic IDE before debugging programs.

  Because the Java virtual machine (JVM) code attaches to the running process, you must exit the Visual Basic editor before debugging your program. If you run the process, then exit your program within the Visual Basic IDE, the JVM code continues to run and you reattach the same JVM code when XJBInit() is called by the debugger. This causes problems if you try to update XJBInit() arguments (for example, classpath) because the changes are not be applied until you restart the Visual Basic program.
- Store the XJB.JClassFactory object globally.

  Because you cannot unload or reinitialize the JVM code, cache the resulting XJB.JClassFactory object as a global variable. The overhead of treating this object as a global variable or passing a single reference around is much less than recreating a new XJB.JClassFactory object and calling the XJBInit() argument more than once.

CScript and Windows Scripting Host

The following guidelines intend to help optimize your use of the ActiveX to EJB bridge with CScript and Windows Scripting Host (WSH):

- Launch in ActiveX to EJB environment.

  Launch the VBScript files in the ActiveX to EJB bridge environment, to run VBScript files in `.vbs` files. Two common ways exist to launch your script:
  - `launchClientXJB MyScript.vbs`
  - `launchClientXJB cscript MyScript.vbs`

Active Server Pages guidelines

The following guidelines intend to help optimize your use of the ActiveX to EJB bridge with Active Server Pages software:

- Use the ActiveX to EJB Helper functions from the Active Server Pages Application.

  Because Active Server Pages (ASP) code typically use VBScript, you can use the included helper functions in any VBScript environment with minor changes. For more information about these helper functions, see Helper functions for data type conversion. To run outside of the ASP environment, remove or change all references to the Server, Request, Response, Application and Session objects; for example, change `Server.CreateObject` to `CreateObject`.
- Set JRE path globally in system.

  The XJB.JClassFactory object must be able to find the Java run time dynamic link library (DLL) when initializing. In Internet Information Server, you cannot specify a path for its processes independently; you must set the process paths in the system PATH variable. You can only have a single JVM version available on a machine using the ASP application. Also, remember that after you change the system PATH variable you must reboot the Internet Information Server machine so that the Internet Information Server can see the change.
- Set the system TEMP environment variable.

  If the system TEMP environment variable is not set, Internet Information Server stores all temporary files in the `WINNT` directory, which is usually not desired.
- Use high isolation or an isolated process.

When using the ActiveX to Java bridge with Active Server Pages software, creating your web application in its own process is recommended. You can only load one JVM instruction in a single process and if you want to have more than one application running with different JVM environment options (for example, different classpaths), then you need to have separate processes.

- Use the Application Unload option.

  When debugging your application, use `Unload` when viewing your ASP application properties in the Internet Information Server administration console to unload the process from memory and thereby unload the JVM code.
- Run one process per application.

  Use only one ASP application per J2EE application or JVM environment, in your ASP environment. If you need separate class paths or JVM settings, you need separate ASP applications (virtual directories with high isolation or an isolated process).
- Store the XJB.JClassFactory object in application scope.

  Because of the one-to-one relationship required between a JVM instruction and a process, and because the JVM code can never detach or shut down from a process independently, cache the XJB.JClassFactory object at application scope and call the XJBInit() method only once.

  Because the ActiveX to EJB bridge employs a free-threaded marshaler, take advantage of the multi-threaded nature of Internet Information Server and the ASP environment. If you choose to reinitialize the XJB.JClassFactory object at Page scope (local variables), then the XJBInit() method can only initialize your local XJB.JClassFactory variable. It is more efficient to use the XJBInit() method once.
- Use VBScript conversion functions.

  Because VBScript code only supports variant data types, use the CStr(), CByte(), CBool(), CCur(), CInt(), Clng(), CSng() and CDbl() functions to tell the activeX to EJB bridge which data type you are using; for example oMyObject.Foo(CDbl(1.234)).

J2EE guidelines

The following guidelines are intended to help optimize your use of the ActiveX to EJB bridge with the J2EE environment;

- Store client container objects globally.

  Because you can only have one JVM instruction per process, and a single J2EE client container (com.ibm.websphere.client.applicationclient.launchClient) per JVM instruction, initialize your J2EE client container only once and reuse it. For ASP applications, store the J2EE client container in an application level variable and initialize it only once (either on the Application_OnStart() event in the `global.asa` file or by checking to see if it IsEmpty()).

  A side effect to storing the client container object globally is that you cannot change the client container parameters without destroying the object and creating a new one. These parameters include the EAR file, BootstrapHost, class path, and so on. If you run a Visual Basic application and want to change the client container parameters, you must end the application and restart it. If you run an Active Server Pages application, you must first unload the application from Internet Information Server (see “Use the Application Unload Button” under Active Server Pages guidelines). Then load the Active Server Pages application with the different client container parameters. The parameters set the first time the Active Server Pages application loads. Since the client container is stored on the Internet Information Server, all the browser clients share the parameters using the Active Server Pages application. This behavior is normal for Active Server Pages code, but can be confusing when you try to run to different WebSphere Application Servers using the same Active Server Pages application, which is not supported.
- Reuse custom temporary directory for EAR file extraction.

  By default, the client container launches and extracts the application `.ear` file to your `temp` directory and then sets up the thread class loader to use the extracted EAR file directory and the JAR files included in the client JAR manifest. This process is time consuming and because of some limitations with JVM shutdown through Java Native Interface (JNI) and file locking, these files are never cleaned up.

  Specifically, each time the client container launch() method is called, it extracts the EAR file to a random directory name in your temporary directory on your hard drive. The current Java thread class loader is

then changed to point to this extracted directory which in turn locks the files within. In a normal J2EE Java client, these files automatically clean up after the application exits. This cleanup occurs when the client container shutdown hook is called (which never happens in the ActiveX to EJB bridge), which leaves the temporary directory there.

To avoid these problems, you can specify a directory to extract the EAR file by setting the com.ibm.websphere.client.applicationclient.archivedir Java system property before calling the client container launch() method. If the directory does not exist or is empty, you extract the EAR file normally. If the EAR file was previously extracted, the directory is reused. This feature is particularly important for server processes (for example, ASP), which can stop and restart, potentially calling the launchClient() method several times.

If you need to update your EAR file, delete the temporary directory first. The next time you create the client container object, it extracts the new EAR file to the temporary directory. If you do not delete the temporary directory or change the system property value to point to a different temporary directory, the client container reuses the currently extracted EAR file, and does not use your changed EAR file.

Note: When specifying the com.ibm.websphere.client.applicationclient.archivedir property, ensure that the directory you specify is unique for each EAR file you use. For example, do not point `MyEar1.ear` and `MyEar2.ear` files to the same directory.

If you choose not to use this system property, go regularly to your Windows `temp` directory and delete the WSTMP* subdirectories. Over a relatively short period of time, these subdirectories can waste a significant amount of space on the hard drive.

ActiveX client programming, tips for passing arrays

Arrays are very similar between Java and Automation containers like Visual Basic and VBScript. This topic provides some important points to consider when passing arrays back and forth between these containers.

Here are some important points to consider when passing arrays back and forth between these containers:

- Java arrays cannot mix types. All Java arrays contain a single type, so when passing arrays of variants to a Java array, you must make sure that all of the elements in the variant array are of the same base type. For example, in Visual Basic code:

```
...
Dim VariantArray(1) as Variant
VariantArray(0) = CLng(123)
VariantArray(1) = CDbl(123.4)
oMyJavaObject.foo(VariantArray) '     Illegal!

VariantArray(0) = CLng(123)
VariantArray(1) = CLng(1234)
oMyJavaObject.foo(VariantArray) '     This works
```

- Arrays of primitive types are converted using the rules defined in primitive data type conversion.
- Arrays of Java objects are handled through arrays of JObjectProxy objects.
- Arrays of JObjectProxy objects must be fully initialized and of the correct associated Java type. When initializing an array in Visual Basic (for example, Dim oJavaObjects(1) as Object), you must set each object to a JObjectProxy object before you send the array to a Java object. The bridge is unable to determine the type of null or empty object values.
- When receiving an array from a Java method, the lower-bound is always zero. Java methods only support zero-based arrays.
- Nested or multidimensional arrays are treated as zero-based multidimensional arrays in Visual Basic and VBScript containers.
- Uninitialized arrays or Array Types are unsupported. When calling a Java method that takes an array of objects as a parameter, you must fully initialize the array of JObjectProxy objects.

ActiveX client programming, Java field programming tips

Using the ActiveX to Enterprise JavaBeans (EJB) bridge to access Java fields has the same case sensitivity issue that it has when invoking methods. Field names must use the same case as the Java field syntax.

Visual Basic code has the same problem with unsolicited case changing on fields as it does with methods. (For more information about this problem, see ActiveX to EJB bridge, calling Java methods). You might use the CallByName() function to set a field in the same way that you call a method in some cases. For fields, use VBLet for primitive types and VBSet for objects. For example:

```
o.MyField = 123                          'Incorrect...
CallByName(o, "MyField", vbLet, 123)     'Correct...
```

or in VBScript:

```
o.MyField = 123                           'Incorrect...
Eval("o.myField = 123")                   'Correct...
```

ActiveX client programming, JClassProxy and JObjectProxy classes

The majority of tasks for accessing your Java classes and objects are handled with the JClassProxy and JObjectProxy objects. This topic provides reference information about the object classes of the ActiveX to Enterprise Java Beans (EJB) bridge.

JClassFactory is the object used to access the majority of Java Virtual Machine (JVM) features. This object handles JVM initialization, accesses classes and creates class instances (objects). Use the JClassProxy and JObjectProxy objects to access the majority of your Java classes and objects:

- XJBInit(String astrJavaParameterArray())

  Initializes the JVM environment using an array of strings that represent the command line parameters you normally send to the `java.exe` file.

  If you have invalid parameters in the XJBInit() string array, the following error is displayed:

  ```
  Error: 0x6002 "XJBJNI::Init() Failed to create VM" when calling XJBInit()
  ```

  If you have C++ logging enabled, the activity log displays the invalid parameter.
- JClassProxy FindClass(String strClassName)

  Uses the current thread class loader to load the specified fully qualified class name and returns a JClassProxy object representing the Java Class object.
- JObjectProxy NewInstance()

  Creates a Class instance for the specified JClassProxy object using the parameters supplied to call the Class constructor. For more information about using the JMethodArgs method, see ActiveX to EJB bridge, calling Java methods.

```
JObjectProxy NewInstance(JClassFactory obj, Variant vArg1, Variant vArg2, Variant vArg3, ...)
JObjectProxy NewInstance(JClassFactory obj, JMethodArgs args)
```

- JMethodArgs GetArgsContainer()

  Returns a JMethodArgs object (Class instance).

  You can create a JClassProxy object from the JClassFactory.FindClass() method and from any Java method call that normally return a Java Class object. You can use this object as if you had direct access to the Java Class object. All of the class static methods and fields are accessible as are the java.lang.Class methods. In case of a clash between static method names of the reflected user class and those of the java.lang.Class (for example, getName()), the reflected static methods would execute first.

  For example, the following is a static method called getName(). The java.lang.Class object also has a method called getName():

  – In Java:

  ```
  class foo{
   foo(){};
   public static String getName(){return "abcdef";}
   public static String getName2(){return "ghijkl";}
   public String toString2(){return "xyz";}
  }
  ```

  – In Visual Basic:

```
...
Dim clsFoo as Object
set clsFoo = oXJB.FindClass("foo")
clsFoo.getName()  ' Returns "abcdef" from the static foo class
clsFoo.getName2() ' Returns "ghijkl" from the static foo class
clsFoo.toString() ' Returns "class foo" from the java.lang.Class object.
oFoo = oXJB.NewInstance(clsFoo)
oFoo.toString() ' Returns some text from the java.lang.Object's
                ' toString() method which foo inherits from.
oFoo.toString2() ' Returns "xyz" from the foo class instance
```

You can create a JObjectProxy object from the JClassFactory.NewInstance() method, and can be created from any Java method call that normally returns a Class instance object. You can use this object as if you had direct access to the Java object and can access all the static methods and fields of the object. All of object instance methods and fields are accessible (including those accessible through inheritance).

The JMethodArgs object is created from the JClassFactory.GetArgsContainer() method. Use this object as a container for method and constructor arguments. You must use this object when overriding the object type when calling a method (for example, when sending a java.lang.String JProxyObject type to a constructor that normally takes a java.lang.Object type).

You can use two groups of methods to add arguments to the collection: Add and Set. You can use Add to add arguments in the order that they are declared. Alternatively, you can use Set to set an argument based on its position in the argument list (where the first argument is in position 1).

For example, if you had a Java Object Foo that took a constructor of Foo (int, String, Object), you could use a JMethodArgs object as shown in the following code extract:

```
...
Dim oArgs as Object
set oArgs = oXJB.GetArgsContainer()

oArgs.AddInt(CLng(12345))
oArgs.AddString("Apples")
oArgs.AddObject("java.lang.Object", oSomeJObjectProxy)

Dim clsFoo as Object
Dim oFoo as Object
set clsFoo = oXJB.FindClass("com.mypackage.foo")
set oFoo = oXJB.NewInstance(clsFoo, oArgs)

' To reuse the oArgs object, just clear it and use the add method
' again, or alternatively, use the Set method to reset the parameters
' Here, we will use Set
oArgs.SetInt(1, CLng(22222))
oArgs.SetString(2, "Bananas")
oArgs.SetObject(3, "java.lang.Object", oSomeOtherJObjectProxy)

Dim oFoo2 as Object
set oFoo2 = oXJB.NewInstance(clsFoo, oArgs)
```

- AddObject (String strObjectTypeName, Object oArg)

  Adds an arbitrary object to the argument container in the next available position, casting the object to the class name specified in the first parameter. Arrays are specified using the traditional [] syntax; for example:

  ```
  AddObject("java.lang.Object[][]", oMy2DArrayOfFooObjects)
  ```

  or

  ```
  AddObject("int[]", oMyArrayOfInts)
  ```

- AddByte (Byte byteArg)

  Adds a primitive byte value to the argument container in the next available position.

- AddBoolean (Boolean bArg)

  Adds a primitive boolean value to the argument container in the next available position.

- AddShort (Integer iArg)

  Adds a primitive short value to the argument container in the next available position.
- AddInt (Long lArg)

  Adds a primitive int value to the argument container in the next available position.
- AddLong (Currency cyArg)

  Adds a primitive long value to the argument container in the next available position.
- AddFloat (Single fArg)

  Adds a primitive float value to the argument container in the next available position.
- AddDouble (Double dArg)

  Adds a primitive double value to the argument container in the next available position.
- AddChar (String strArg)

  Adds a primitive char value to the argument container in the next available position.
- AddString (String strArg)

  Adds the argument in string form to the argument container in the next available position.
- SetObject (Integer iArgPosition, String strObjectTypeName, Object oArg)

  Adds an arbitrary object to the argument container in the specified position casting it to the class name or primitive type name specified in the second parameter. Arrays are specified using the traditional [] syntax; for example:

  ```
  SetObject(1, "java.lang.Object[][]", oMy2DArrayOfFooObjects)
  ```

  or

  ```
  SetObject(2, "int[]", MyArrayOfInts)
  ```
- SetByte (Integer iArgPosition, Byte byteArg)

  Sets a primitive byte value to the argument container in the position specified.
- SetBoolean (Integer iArgPosition, Boolean bArg)

  Sets a primitive boolean value to the argument container in the position specified.
- SetShort (Integer iArgPosition, Integer iArg)

  Sets a primitive short value to the argument container in the position specified.
- SetInt (Integer iArgPosition, Long lArg)

  Sets a primitive int value to the argument container in the position specified.
- SetLong (Integer iArgPosition, Currency cyArg)

  Sets a primitive long value to the argument container in the position specified.
- SetFloat (Integer iArgPosition, Single fArg)

  Sets a primitive float value to the argument container in the position specified.
- SetDouble (Integer iArgPosition, Double dArg)

  Sets a primitive double value to the argument container in the position specified.
- SetChar (Integer iArgPosition, String strArg)

  Sets a primitive char value to the argument container in the position specified.
- SetString (Integer iArgPosition, String strArg)

  Sets a java.lang.String value to the argument container in the position specified.
- Object Item(Integer iArgPosition)

  Returns the value of an argument at a specific argument position.
- Clear()

  Removes all arguments from the container and resets the next available position to one.
- Long Count()

  Returns the number of arguments in the container.

ActiveX client programming, Java virtual machine initialization tips

Initialize the Java virtual machine (JVM) code with the ActiveX to Enterprise Java Beans (EJB) bridge. For an ActiveX client program (Visual Basic, VBScript, or ASP) to access Java classes or objects, the first step that the program must do is to create Java virtual machine (JVM) code within its process.

To create JVM code, the ActiveX program calls the XJBInit() method of the XJB.JClassFactory object. When an XJB.JClassFactory object is created and the XJBInit() method called, the JVM is initialized and ready to use.

- To enable the XJB.JClassFactory to find the Java run-time description definition language (DLL) when initializing, the Java Runtime Environment (JRE) bin and `bin\classic` directories must exist in the system path environment variable.
- The XJBInit() method accepts only one parameter: an array of strings. Each string in the array represents a command line argument that for a Java program you would normally specify on the Java.exe command line. This string interface is used to set the class path, stack size, heap size and debug settings. You can get a listing of these parameters by typing `java -?` from the command line.
- If you set a parameter incorrectly, you receive a 0x6002 “Failed to initialize VM” error message.
- Due to the current limitations of Java Native Interface (JNI), you cannot unload or reinitialize the JVM code after it has loaded. Therefore, after the XJBInit() method has been called once, subsequent calls have no effect other than to create a duplicate JClassFactory object for you to access. It is best to store your XJB.JClassFactory object globally and continue to reuse that object.
- The following Visual Basic extract shows an example of initializing JVM code:

```
Dim oXJB as Object
set oXJB = CreateObject("XJB.JClassFactory")
Dim astrJavaInitProps(0) as String
astrJavaInitProps(0) = _
     "-Djava.class.path=.;c:\myjavaclasses;c:\myjars\myjar.jar"
oXJB.XJBInit(astrJavaInitProps)
```

ActiveX to Java primitive data type conversion values

All primitive Java data types are automatically converted to native ActiveX Automation types. However, not all Automation data types are converted to Java types (for example, VT_DATE). Variant data types are used for data conversion.

Variant data types are a requirement of any Automation interface, and are used automatically by Visual Basic and VBScript. The following tables provide details about how primitive data types are converted between Automation types and Java types.

Table 4. ActiveX to Java primitive data type conversion. Conversion details for primitive data types

Visual Basic Type	Variant Type	Java Type	Notes
Byte	VT_I1	byte	Byte in Visual Basic is unsigned, but is signed in Java data type.
Boolean	VT_BOOL	boolean	
Integer	VT_I2	short	
Long	VT_I4	int	
Currency	VT_CY	long	
Single	VT_R4	float	
Double	VT_R8	double	
String	VT_BSTR	java.lang.String	
String	VT_BSTR	char	
Date	VT_DATE	n/a	

Example: ActiveX client application using helper methods for data type conversion:

Generally, data type conversion between ActiveX (Visual Basic and VBScript) and Java methods occurs automatically, as described in ActiveX to EJB bridge, converting data types. However, the byte helper function and currency helper function are provided for cases where automatic conversion is not possible:

- Byte helper function

Because the Java Byte data type is signed (-127 through 128) and the Visual Basic Byte data type is unsigned (0 through 255), convert unsigned Bytes to a Visual Basic Integers, which look like the Java signed byte. To make this conversion, you can use the following helper function:

```
Private Function GetIntFromJavaByte(Byte jByte) as Integer
      GetIntFromJavaByte = (CInt(jByte) + 128) Mod 256 - 128
End Function
```

- Currency helper function

  Visual Basic 6.0 cannot properly handle 64-bit integers like Java methods can (as the Long data type). Therefore, Visual Basic uses the Currency type, which is intrinsically a 64-bit data type. The only side effect of using the Currency type (the Variant type VT_CY) is that a decimal point is inserted into the type. To extract and manipulate the 64-bit Long value in Visual Basic, use code like the following example. For more details on this technique for converting Currency data types, see “Q189862, HOWTO: Do 64-bit Arithmetic in VBA”, on the Microsoft Knowledge Base.

```
' Currency Helper Types
Private Type MungeCurr
    Value As Currency
End Type
Private Type Munge2Long
    LoValue As Long
    HiValue As Long
End Type

' Currency Helper Functions
Private Function CurrToText(ByVal Value As Currency) As String
    Dim Temp As String, L As Long
    Temp = Format$(Value, "#.0000")
    L = Len(Temp)
    Temp = Left$(Temp, L - 5) & Right$(Temp, 4)
    Do While Len(Temp) > 1 And Left$(Temp, 1) = "0"
        Temp = Mid$(Temp, 2)
    Loop
    Do While Len(Temp) > 2 And Left$(Temp, 2) = "-0"
        Temp = "-" & Mid$(Temp, 3)
    Loop
    CurrToText = Temp
End Function

Private Function TextToCurr(ByVal Value As String) As Currency
    Dim L As Long, Negative As Boolean
    Value = Trim$(Value)
    If Left$(Value, 1) = "-" Then
        Negative = True
        Value = Mid$(Value, 2)
    End If
    L = Len(Value)
    If L < 4 Then
        TextToCurr = CCur(IIf(Negative, "-0.", "0.") & _
                          Right$("0000" & Value, 4))
    Else
        TextToCurr = CCur(IIf(Negative, "-", "") & _
                          Left$(Value, L - 4) & "." & Right$(Value, 4))
    End If
End Function

' Java Long as Currency Usage Example
Dim LC As MungeCurr
Dim L2 As Munge2Long

' Assign a Currency Value (really a Java Long)
' to the MungeCurr type variable
LC.Value = cyTestIn
```

```
' Coerce the value to the Munge2Long type variable
LSet L2 = LC

' Perform some operation on the value, now that we
' have it available in two 32-bit chunks
L2.LoValue = L2.LoValue + 1

' Coerce the Munge value back into a currency value
LSet LC = L2
cyTestIn = LC.Value
```

ActiveX client programming, handling error codes

All exceptions thrown in Java code are encapsulated and thrown again as a COM error through the ISupportErrorInfo interface and the EXCEPINFO structure of IDispatch::Invoke(), the Err object in Visual Basic and VBScript. Because there are no error numbers associated with Java exceptions, whenever a Java exception is thrown, the entire stack trace is stored in the error description text and the error number assigned is 0x6003.

In Visual Basic or VBScript, you need to use the Err.Number and Err.Description fields to determine the actual Java error. Non-Java errors are thrown as you would expect via the IDispatch interface; for example, if a method cannot be found, then error 438 “Object doesn't support this property or method” is thrown.

Table 5. Error numbers and descriptions.. Error numbers and descriptions in VBScript

Error number	Description
0x6001	Java Native Interface (JNI) error
0x6002	Initialization error
0x6003	Java exception. Error description is the Java Stack Trace.
0x6FFF	General Internal Failure

ActiveX client programming, threading tips

The ActiveX to Enterprise JavaBeans (EJB) bridge supports both free-threaded and apartment-threaded access and implements the Free Threaded Marshaler to work in a hybrid environment such as Active Server Pages (ASP). Each thread created in the ActiveX process is mirrored in the Java environment when the thread communicates through the ActiveX to EJB bridge.

When all references to Java objects (there are no JObjectProxy or JClassProxy objects) are loaded in an ActiveX thread, the ActiveX to EJB bridge detaches the thread from the Java virtual machine (JVM) code. Therefore, you must be careful that any Java code that you access from a multithreaded Windows application is thread safe. Visual Basic code and VBScript applications are both essentially single threaded. Therefore, Visual Basic and VBScript applications do not have threading issues in the Java programs they access. Active Server Pages and multithreaded C and C++ programs can have issues.

Consider the following scenario:

1. A multithreaded Windows Automation Container (our ActiveX Process) starts. It exists on Thread A.
2. The ActiveX Process initializes the ActiveX to EJB bridge, which starts the JVM code. The JVM attaches to the same thread and internally calls it Thread 1.
3. The ActiveX Process starts two threads: B and C.
4. Thread B in the ActiveX Process uses the ActiveX to EJB bridge to access an object that was created in Thread A. The JVM attaches to thread B and calls it Thread 2.
5. Thread C in the ActiveX Process never talks to the JVM code, so the JVM never needs to attach to it. This is a case where the JVM code does not have a one-to-one relationship between ActiveX threads and Java threads.
6. Thread B later releases all of the JObjectProxy and JClassProxy objects that it used. The Java Thread 2 is detached.

7. Thread B again uses the ActiveX to EJB bridge to access an object that was created in Thread A. The JVM code attaches again to the thread and calls it Thread 3.

Table 6. Thread scenario.. Thread scenario

ActiveX process	JVM access by ActiveX process
Thread A - Created in 1	Thread 1 - Attached in 2
Thread B - Created in 4	Thread 2 - Attached in 4, detached in 6 Thread 3 - Attached in 7
Thread C - Created in 4	

Threads and Active Server Pages

Active Server Pages (ASP) in Microsoft Internet Information Server is a multithreaded environment. When you create the XJB.JClassFactory object, you can store it in the Application collection as an Application-global object. All threads within your ASP environment can now access the same ActiveX to EJB bridge object. Active Server Pages by default creates 10 Apartment Threads per ASP process per CPU. This means that when your ActiveX to EJB bridge object is initialized any of the 10 threads can call this object, not just the thread that created it.

If you need to simulate single-apartment behavior, you can create a Single-Apartment Threaded ActiveX dynamic link library (DLL) in Visual Basic code and encapsulate the ActiveX to the EJB bridge object. This encapsulation guarantees that all access to the JVM object is on the same thread. You need to use the <OBJECT> tag to assign the XJB.JClassFactory to an Application object and must be aware of the consequences of introducing single-threaded behavior to a web application.

The Microsoft KnowlegeBase has several articles about ASP and threads, including:
- Q243543 INFO: Do Not Store STA Objects in Session or Application
- Q243544 INFO: Component Threading Model Summary Under Active Server Pages
- Q243548 INFO: Design Guidelines for VB Components Under ASP

Example: Enabling logging and tracing for activeX client applications

The ActiveX to EJB bridge provides two logging and tracing formats: Windows Application Event Log and Java Trace Log.

- Windows Event Log

  The Windows Application Event Log shows JNI errors, Java console error messages, and XJB initialization messages. This log is most useful for determining XJBInit() errors and any unusual exceptions that do not come from the Java environment. By default, critical error logging will be enabled and debug and event logging is disabled.

  To enable or disable logging of certain event types to the Windows Event Log, specify one or more parameters to XJBInit(). If more than one parameter is set, they will be processed in the order in which they appear in the input string array to the XJBInit() method. Once the XJBInit() method is initialized, these parameters can no longer be set/reset for the life of the process. Using Java java.lang.System.setProperty() to set these values also has no effect.
  - -Dcom.ibm.ws.client.xjb.native.logging.debug=enabled|disabled

    Enables or disables debug level messages from displaying in the Windows operating system event log. This level of logging is most useful and shows most internal errors, user programming issues or configuration problems.
  - -Dcom.ibm.ws.client.xjb.native.logging.event=enabled|disabled

    Enables or disables event level messages from appearing in the Windows operating system event log.
  - -Dcom.ibm.ws.client.xjb.native.logging.*=enabled|disabled

Enables or disables both event and debug level messages from appearing in the Windows operating system event log. It is not possible to disable some critical error messages from being displayed in the error log. Only debug and event level messages can be disabled.

To view the Windows application event log with the Event Viewer, complete the following steps:

1. Click **Start** > **Control Panel**.
2. Double-click **Administrative Tools**.
3. Double-click **Event Viewer**.

All ActiveX to EJB bridge events display the text `WebSphere XJB` in the source column and in the application log.

When using the Event Viewer, you can get help information by clicking the menu choice **Action** > **Help**.

- Java trace log

  The Java trace log displays information that you can use to debug method calls, class lookups, and argument coercion problems. Because the Java portion of the bridge mirrors the function of the COM IDispatch interface, the information in the trace log is similar to what you have come to expect from an IDispatch interface. To understand the trace log, you need a fundamental understanding of IDispatch.

  To enable user-logging, add the following parameters to the XJBInit() input string array:

```
"-DtraceString=com.ibm.ws.client.xjb.*=event=enabled"
"-DtraceFile=C:\MyTrace.txt"
```

Example: Viewing a System.out message

The ActiveX to Enterprise JavaBeans (EJB) bridge does not have a console available to view Java System.out messages. To view these messages when running a stand-alone client program (such as Visual Basic), redirect the output to a file.

This example redirects output to a file:

```
launchClientXJB.bat MyProgram.exe > output.txt
```

- To view the System.out messages when running a Service program such as Active Server Pages, you need to override the Java System.out OutputStream object to FileOutputStream. For example, in VBScript:

```
'Redirect system.out to a file
' Assume that oXJB is an initialized XJB.JClassFactory object
 Dim clsSystem
 Dim oOS
 Dim oPS
 Dim oArgs

' Get the System class
  Set clsSystem = oXJB.FindClass("java.lang.System")

' Create a FileOutputStream object
' Create a PrintStream object and assign to it our FileOutputStream
  Set oArgs = oXJB.GetArgsContainer  oArgs.AddObject "java.io.OutputStream", oOS
  Set oPS = oXJB.NewInstance(oXJB.FindClass("java.io.PrintStream"), oArgs)

' Set our System OutputStream to our file
  clsSystem.setOut oPS
```

Developing applet client code

Applet clients are capable of communicating over the HTTP protocol and the RMI-IIOP protocol.

About this task

Unlike typical applets that reside on either web servers or WebSphere Application Servers and can only communicate using the HTTP protocol, applet clients are capable of communicating over the HTTP protocol and the RMI-IIOP protocol. This additional capability gives the applet direct access to enterprise beans.

Standard applets require the HTML <APPLET> tag to identify the applet to the browser. If you replace the <OBJECT> and <EMBED> tags, make sure that you specify appropriate values for <OBJECT> and <EMBED> tags, especially the <OBJECT classid and <EMBED type values.

In the code for your applet client, when you initialize an instance of the InitialContext class, you must set properties to specify the computer name, domain, and port, and to identify this program (the client) as an applet.

Example

- Applet client tag requirements
- Applet client code requirements

Applet client tag requirements: Standard applets require the HTML <APPLET> tag to identify the applet to the browser. The <APPLET> tag invokes the Java virtual machine (JVM) of the browser. It can also be replaced by <OBJECT> and <EMBED> tags. The following code example illustrates the applet code using the <APPLET> tag:

```
<APPLET code="MyAppletClass.class" archive="Applet.jar, EJB.jar" width="600" height="500" >
</APPLET>
```

The following code example illustrates the applet code using the <OBJECT> and <EMBED> tags.

```
<OBJECT classid="clsid: 8AD9C840-044E-11D1-B3E9-00805F499D93"
width="600" height="500">
<PARAM NAME=CODE VALUE=MyAppletClass.class>
<PARAM NAME="archive" VALUE='Applet.jar, EJB.jar'>
<PARAM TYPE="application/x-java-applet;version=1.5.0">
<PARAM NAME="scriptable" VALUE="false">
<PARAM NAME="cache-option" VALUE="Plugin">
<PARAM NAME="cache-archive" VALUE="Applet.jar, EJB.jar">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.5.0" CODE=MyAppletClass.class
ARCHIVE="Applet.jar, EJB.jar" WIDTH="600" HEIGHT="500"
scriptable="false">
<NOEMBED>
</COMMENT>
</NOEMBED>WebSphere Java Application/Applet Thin Client for
Windows is required.
</EMBED>
</OBJECT>
```

Attention: To successfully invoke the applet client in WebSphere Application Server version 6.1 or later, the <OBJECT classid and <EMBED type values need to be those shown in the preceding example.

For more information about the <APPLET> tag, see the article, Using applet, object and embed Tags.

Applet client code requirements: The code used by an applet to talk to an enterprise bean is the same as that used by a stand-alone Java program or a servlet, except for one additional property called java.naming.applet. This property informs the InitialContext and the Object Request Broker (ORB) that this client is an applet rather than a stand-alone Java application or servlet. The following code example illustrates the applet code using the <APPLET> tag:

When you initialize an instance of the InitialContext class, the first two lines in this code snippet illustrate what both a stand-alone Java program and a servlet issue to specify the computer name, domain, and port. In this example, <yourserver.yourdomain.com> is the computer name and domain where WebSphere Application Server resides, and 900 is the configured port. After the bootstrap values (<yourserver.yourdomain.com>:900) are defined, the client to server communications occur within the underlying infrastructure. In addition to the first two lines for applets, you must add the third line to your code, which identifies this program as an applet, for example:

```
prop.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
prop.put(Context.PROVIDER_URL, "iiop://<yourserver.yourdomain.com>:900)
prop.put(Context.APPLET, this);
```

Example: Applet client tag requirements

Standard applets require the HTML <APPLET> tag to identify the applet to the browser. The <APPLET> tag invokes the Java virtual machine (JVM) of the browser. It can also be replaced by <OBJECT> and <EMBED> tags.

The following code example illustrates the applet code using the <APPLET> tag.

```
<APPLET code="MyAppletClass.class" archive="Applet.jar, EJB.jar" width="600" height="500" >
</APPLET>
```

The following code example illustrates the applet code using the <OBJECT> and <EMBED> tags.

```
<OBJECT classid="clsid: 8AD9C840-044E-11D1-B3E9-00805F499D93"
width="600" height="500">
<PARAM NAME=CODE VALUE=MyAppletClass.class>
<PARAM NAME="archive" VALUE='Applet.jar, EJB.jar'>
<PARAM TYPE="application/x-java-applet;version=1.5.0">
<PARAM NAME="scriptable" VALUE="false">
<PARAM NAME="cache-option" VALUE="Plugin">
<PARAM NAME="cache-archive" VALUE="Applet.jar, EJB.jar">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.5.0" CODE=MyAppletClass.class
ARCHIVE="Applet.jar, EJB.jar" WIDTH="600" HEIGHT="500"
scriptable="false">
<NOEMBED>
</COMMENT>
</NOEMBED>WebSphere Java Application/Applet Thin Client for
Windows is required.
</EMBED>
</OBJECT>
```

Attention: To successfully invoke the applet client in WebSphere Application Server version 6.1 or later, the <OBJECT classid and <EMBED type values need to be those shown in the preceding example.

For more information about the <APPLET> tag, see Using applet, object and embed Tags.

Example: Applet client code requirements

The code used by an applet to talk to an enterprise bean is the same as that used by a stand-alone Java program or a servlet, except for one additional property called java.naming.applet. This property informs the InitialContext and the Object Request Broker (ORB) that this client is an applet rather than a stand-alone Java application or servlet.

When you initialize an instance of the InitialContext class, the first two lines in this code snippet illustrate what both a stand-alone Java program and a servlet issue to specify the computer name, domain, and port. In this example, <yourserver.yourdomain.com> is the computer name and domain where WebSphere Application Server resides, and 900 is the configured port. After the bootstrap values (<yourserver.yourdomain.com>:900) are defined, the client to server communications occur within the underlying infrastructure. In addition to the first two lines for applets, you must add the third line to your code, which identifies this program as an applet, for example:

```
prop.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
prop.put(Context.PROVIDER_URL, "iiop://<yourserver.yourdomain.com>:900)
prop.put(Context.APPLET, this);
```

Example: Enabling logging and tracing for application clients

The ActiveX to EJB bridge provides two logging and tracing formats: Windows Application Event Log and Java Trace Log.

- Windows Event Log

The Windows Application Event Log shows JNI errors, Java console error messages, and XJB initialization messages. This log is most useful for determining XJBInit() errors and any unusual exceptions that do not come from the Java environment. By default, critical error logging will be enabled and debug and event logging is disabled.

To enable or disable logging of certain event types to the Windows Event Log, specify one or more parameters to XJBInit(). If more than one parameter is set, they will be processed in the order in which they appear in the input string array to the XJBInit() method. Once the XJBInit() method is initialized, these parameters can no longer be set/reset for the life of the process. Using Java `java.lang.System.setProperty()` to set these values also has no effect.

- `-Dcom.ibm.ws.client.xjb.native.logging.debug=enabled|disabled`

  Enables or disables debug level messages from displaying in the Windows operating system event log. This level of logging is most useful and shows most internal errors, user programming issues or configuration problems.
- `-Dcom.ibm.ws.client.xjb.native.logging.event=enabled|disabled`

  Enables or disables event level messages from appearing in the Windows operating system event log.
- `-Dcom.ibm.ws.client.xjb.native.logging.*=enabled|disabled`

  Enables or disables both event and debug level messages from appearing in the Windows operating system event log. It is not possible to disable some critical error messages from being displayed in the error log. Only debug and event level messages can be disabled.

Viewing the Windows application event log with the event viewer:

To open the event viewer in the Windows operating system:

1. Click **Start** > **Settings** > **Control Panel**.
2. Double-click Administrative Tools.
3. Double-click Event Viewer.

All ActiveX to EJB bridge events display the text WebSphere XJB in the source column and in the application log. For information about using Event Viewer, click the **Action** menu in Event Viewer, and then click **Help**.

To open the even viewer in the Windows operating system, click **Start** > **Programs** > **Administrative Tools** > **Event Viewer**. All ActiveX to EJB bridge events have the text WebSphere XJB in the source column and display in the application log. For information about using Event Viewer, click the **Help** menu in Event Viewer.

- Java trace log

  The Java trace log displays information that you can use to debug method calls, class lookups, and argument coercion problems. Since the Java portion of the bridge mirrors the function of the COM IDispatch interface, the information in the trace log is similar to what you have come to expect from an IDispatch interface. To understand the trace log, you need a fundamental understanding of IDispatch.

  To enable user-logging, add the following parameters to the XJBInit() input string array:

```
"-DtraceString=com.ibm.ws.client.xjb.*=event=enabled"
"-DtraceFile=C:\MyTrace.txt"
```

Chapter 6. Developing Communications Enabled Applications

Communications Enabled Applications (CEA) is a functionality that provides the ability to add dynamic web communications to any application or business process. The product provides a suite of integrated telephony and collaborative web services that extends the interactivity of enterprise and web commerce applications. With the CEA capability, enterprise solution architects and developers can use a single core application to enable multiple modes of communication. Enterprise developers do not need to have extensive knowledge of telephony or Session Initiation Protocol (SIP) to implement CEA. The CEA capability delivers call control, notifications, and interactivity and provides the platform for more complex communications.

Developing communications enabled applications

Developing SIP communications applications

Before you begin

Note: Failover for SIP applications is supported on the z/OS platform.

Procedure

- Use the Session Initiation Protocol (SIP) application router to select the order in which SIP applications are triggered.

  When configuring a SIP application router, you can either use the default application router or create a custom application router.
- Use the Asynchronous Invocation API to transfer events that require processing in the context of a Session Initiation Protocol (SIP) application session to any server in a cluster based on the related application session ID.

  The Asynchronous Invocation API transfers the event task to the correct server.
- Use the Domain Resolver API to perform DNS lookups of SIP URIs using the RFC 3263 protocol.

  The Domain Resolver API provides an interface that enables an application to perform DNS queries for SIP URIs.
- Create a SIP servlet request that includes proprietary header fields.

  SIP proprietary header fields enable certain SIP settings to be implemented on a per message basis. SIP settings set at the SIP container level apply to all SIP messages handled by that SIP container.

Domain Resolver API

Use the Domain Resolver API to perform DNS lookups of SIP URIs using the RFC 3263 protocol. These lookups can be performed synchronously if you want to avoid having to preserve state in order to handle an asynchronous callback. They can also be performed asynchronously if you need a better performing interface.

The Domain Resolver API provides an interface that enables an application to perform DNS queries for SIP URIs. When the Domain Resolver API is used asynchronously, a listener is called after the DNS query completes. You can use one of the following methods in your application code to access the Domain Resolver API:

- Get an attribute from the ServletContext using com.ibm.ws.sip.container.domain.resolver as a key.

  ```
  getServletContext().getAttribute("com.ibm.ws.sip.container.domain.resolver")
  ```
- Use @resource injection.

  ```
  @resource
  DomainResolver resolver
  ```

Use one of the following methods to perform the URI lookup:

- Use the SIPURI method if a synchronous API which will return the result of the URI resolve request response.

```
DomainResolver
locate(SIPURI)
```

- locate(SIPURI, Listener) – an asynchronous API which will signal the listener once it is finished. When the result is cached the listener will be triggered on the same caller thread.

```
DomainResolver
locate(SIPURI, Listener)
```

For more information about this API, expand the **Reference > Programming interfaces > APIs - Application Programming Interfaces** section in the Feature Pack for Communications Enabled Applications Information Center navigation. Then scroll down to the com.ibm.ws.sip.container.domain.resolver API.

For more information about DNS servers, see the topic *Using DNS procedures to locate SIP servers*

SIP proprietary header fields

You can create a SIP servlet request that includes proprietary header fields. SIP proprietary header fields enable certain SIP settings to be implemented on a per message basis. SIP settings set at the SIP container level apply to all SIP messages handled by that SIP container.

To include one or more proprietary header fields in a message, set up your SIP servlet request such that it includes one or more SipServletMessage.setHeader(*string_name*, *string value*) methods. When the application calls SipServletRequest.send() to send the request, the message object that is passed to the SIP stack for transmission includes the propriety header information. The SIP stack then creates a client transaction to send out the request, and adjusts the SIP configuration settings for this specific request based on any proprietary header fields that are included in the message object. The stack removes the proprietary header fields before the message is sent out to the network.

Proprietary header fields used to specify timer values

The following proprietary header fields are available for specifying timer values for a specific message. The application can set multiple timer values in one message instance, but cannot specify multiple values for the same proprietary header field.

IBM-TransactionTimeout
Use this header field to specify, in milliseconds, the length of the client transaction timeout. This header is equivalent to specifying a value for timer B in INVITE client transactions, and timer F in non-INVITE client transactions.

IBM-RetransmissionInterval
Use this header field to specify, in milliseconds, the length of the request retransmission interval. This header is equivalent to specifying a value for timer A in INVITE client transactions, and timer E in non-INVITE client transactions.

IBM-RetransmissionMaxInterval
Use this header field to specify, in milliseconds, the maximum retransmission interval. This header is equivalent to specifying a value for timer T2 in non-INVITE client transactions, and timer B in INVITE client transactions.

Chapter 7. Developing data access resources

This page provides a starting point for finding information about data access. Various enterprise information systems (EIS) use different methods for storing data. These backend data stores might be relational databases, procedural transaction programs, or object-oriented databases.

The flexible IBM WebSphere Application Server provides several options for accessing an information system backend data store:

- Programming directly to the database through the JDBC 4.0 API, JDBC 3.0 API, or JDBC 2.0 optional package API.
- Programming to the procedural backend transaction through various J2EE Connector Architecture (JCA) 1.0 or 1.5 compliant connectors.
- Programming in the bean-managed persistence (BMP) bean or servlets indirectly accessing the backend store through either the JDBC API or JCA-compliant connectors.
- Using container-managed persistence (CMP) beans.
- Using the IBM data access beans, which also use the JDBC API, but give you a rich set of features and function that hide much of the complexity associated with accessing relational databases.

Service Data Objects (SDO) simplify the programmer experience with a universal abstraction for messages and data, whether the programmer thinks of data in terms of XML documents or Java objects. For programmers, SDOs eliminate the complexity of the underlying data access technology such as, JDBC, RMI/IIOP, JAX-RPC, and JMS, and message transport technology such as, java.io.Serializable, DOM Objects, SOAP, and JMS.

Developing data access applications

Developing data access applications

You can use data access applications to manipulate data from outside sources for use within your application serving environment.

About this task

You can access data in various ways:

- using standard or extended APIs
- using container-managed persistence beans
- using bean-managed persistence beans, session beans, or web components.
- using Service Data Objects (SDO)

Procedure

1. Decide how to implement data access.

   The Enterprise JavaBeans (EJB) programming model provides several distinct server-side component types: entity, session, and message-driven beans, and servlets. Of these types, entity beans are typically used to model business components in an application. Entity beans have both *state* and *behavior*.

   The state of entity beans is persistent and is stored in a database. As changes are made to an entity bean, its state is kept in synchronization with the database record representing the bean. There are two types of entity beans provided by the EJB model and these two types differ in the mechanism used to provide persistence. These two types of entity beans are *container-managed persistence* (CMP) beans and *bean-managed persistence* (BMP) beans.

   - With BMP beans, the developer manually produces code to manage the persistent state of the bean.

- With CMP beans, the EJB container manages the persistent state of the bean. Persistent state management is a complex and difficult task; using CMP beans allows the developer to concentrate on business logic by delegating persistence behavior to the container.

  Typical examples of CMP beans are *Customer*, *Account*, and so on. Because CMP beans are objects, their data (state) is accessed using field accessors. For example, a *Customer* entity bean is likely to have fields such as *name* and *phoneNumber*. These pieces of data are accessed using the accessor methods *getName()/setName()* and *getPhoneNumber()/setPhoneNumber()*. As a developer, you are not concerned with how this data is eventually stored and retrieved from the backend database and can assume that the integrity of the data is maintained by the container.

See the topic, Developing enterprise beans for information on developing entity beans.

Tips:

- To maximize the efficiency of application requests to relational databases, consider using Structured Query Language in Java (SQLJ) when developing BMP and CMP beans. This option is available for applications that use the DB2 JDBC Universal Driver to access DB2 databases.

  The only exception to this driver requirement applies to SQLJ-backed BMP beans that access DB2 for z/OS; this schema requires the DB2 for z/OS Legacy Driver (required for the DB2 for z/OS Local JDBC Provider RRS).
- Also consider using cursor holdability for potential performance gains; see the topic, JDBC application cursor holdability support, for details.

An alternative to developing entity beans is using the Service Data Objects (SDO) framework, which is a unified framework for data application development. With SDO, you do not need to be familiar with a technology-specific API in order to access and utilize data. You need to know only one API, the SDO API, which lets you work with data from multiple sources, including relational databases, entity EJB components, XML pages, web services, the Java Connector Architecture, JavaServer Pages, and more.

2. Look up a data source or connection factory using a resource reference. See the topic, Looking up data sources with resource references for relational access for more information. *Do not perform this step if you work with CMP beans. The EJB container handles this process for CMP beans.*

   To run applications on WebSphere Application Server, your code must use resource references to logically named data sources or connection factories. Mapping the resource references to actual resources is usually done at assembly time. The Application Server administrator configures those resources.

   - For relational database access, administrators configure a JDBC provider and associated data sources, which work with the embedded WebSphere Relational Resource Adapter.
   - For non-relational database access, administrators install a Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) resource adapter onto an application server and configure associated connection factories.

   Generic work context implementation provides a mechanism for a resource adapter to control the contexts in which instances of work submitted by the resource adapter to the product work manager for execution are executed. By submitting a work instance that implements the WorkContextProvider interface, the resource adapter can propagate various types of context to the WebSphere Application Server. The application server then, if it supports the propagated context type, sets the provided context as the execution context of the work instance during its execution.
3. Get a connection to a data source or a connection factory. See the "Getting connections" section of the topic, Connection life cycle for details.) *Do not perform this step if you work with CMP beans. The EJB container handles this process for CMP beans.*

   The connection management architecture for both relational and procedural access to enterprise information systems (EIS) is based on the Java EE Connector Architecture (JCA) specification. The Connection Manager (CM), which pools and manages connections within an application server, is capable of managing connections obtained through both resource adapters (RAs) defined by the JCA specification, and data sources defined by the JDBC Extensions Specification.

4. Use thread identity to assign an owner to the connection. See the topic, Using thread identity, support for more information.

Using Bean Validation in RAR modules

WebSphere Application Server validates resource archive (RAR) JavaBeans constraints in compliance with the Java Connector Architecture (JCA) version 1.6 specification.

Bean validation in RAR modules:

WebSphere Application Server validates resource adapter archive (RAR) JavaBeans constraints in compliance with the Java Connector Architecture (JCA) version 1.6 specification.

Resource adapters can specify the validation requirements of configuration properties to the Application Server through annotations in the source code of the resource adapter, constraint specifications in a resource adapter validation descriptor, or a mixture of both. In specifying these constraints, resource adapters can use the built-in bean validation constraints supplied with the Application Server, custom bean validation constraints supplied either by the application developer or a third party, or a mixture of both. Resource adapter developers can apply constraints to the fields and JavaBeans-compliant properties of the following JCA types:

- ResourceAdapter
- ManagedConnectionFactory
- ActivationSpec
- AdministeredObject

At run time, the application server creates instances of bean types declared by the resource adapter. Each instance is validated immediately upon setting its configuration properties, before placing the instance into service.

When validating a RAR bean, the Application Server creates an instance of a validator factory according to the bean validation deployment descriptor discovered by the Application Server. A validator instance is then obtained from the factory and used to validate the bean instance.

If validation fails, the Application Server throws a constraint violation exception and reports all violations to the system log. The effects of the exception for each RAR bean type and problem determination information are documented in the topic, Troubleshooting bean validation in RAR modules.

Note: The Bean Validation specification requires that no more than one `validation.xml` is visible on the class path. This requirement is violated whenever two or more stand-alone RARs provide a validation descriptor. See the section, “RAR bean validation descriptor” in this topic, for more information. When more than one `validation.xml` is visible to the Application Server class loaders, the Application Server or application modules might fail to acquire the default ValidatorFactory and subsequently cannot perform bean validation. For example, the server cannot validate beans of a RAR embedded in an application whenever the embedded RAR lacks a validation configuration, and two or more stand-alone RARs provide configurations. To avoid trouble, install stand-alone RARs that provide a bean validation descriptor as isolated whenever possible.

Built-in constraint annotations

Note: Use built-in constraint annotations to specify the range and mandatory attributes of configuration properties rather than provide custom annotations for the same purpose. The following constraints are useful, but you can use all bean validation built-in constraints. See the topic Bean validation built-in constraints for a complete list of the constraints.

- @Min

  Specifies the minimum value of the configuration property decorated with this annotation. The value must be greater than or equal to the specified minimum.

- @Max

  Specifies the maximum value of the configuration property decorated with this annotation. The value must less than or equal to the specified maximum.
- @Size

  Specifies the range of values of the configuration property decorated with this annotation. The value must be greater than or equal to the specified minimum and be less than or equal to the specified maximum.
- @NotNull

  Specifies the value of the configuration property decorated with this annotation must not be null. That is, the property is required.

The following example is a RAR bean class that is decorated with built-in constraint annotations.

The value of the serverName configuration property must not be null, and the value of the instanceCount property must be at least 1 when the Application Server creates and configures an instance of the MyConnector class. Otherwise, a constraint validation exception occurs and, in the case of ResourceAdapter bean, the resource adapter fails to start. See the topic Troubleshooting bean validation in RAR modules for more information.

```
package com.my.company;

@Connector(...)
public class MyConnector implements ResourceAdapter, Serializable
{
 @ConfigProperty(type=java.lang.String.class,defaultValue="WAS")
 private String serverName;

 @NotNull()
 public String getServerName() {return serverName;}

 private Integer instanceCount = 0;

 @Min(value=1)
 public Integer getInstanceCount() {return instanceCount;}
 ...
```

RAR bean validation descriptor

Bean validation constraints can be declared through an XML descriptor supplied by a RAR module. In the simplest case, a RAR validation descriptor consists of the validation configuration declared in the validation.xml file and zero or more XML files that declare RAR bean validation constraints. Files containing constraint declarations are specified in the constraint-mapping elements of the validation configuration (validation.xml).

You must package the validation descriptor in the META-INF directory of a RAR module. Any custom constraint annotation classes that are declared in the validation descriptor must also be packaged in the RAR module.

The following example is a simple RAR validation descriptor that declares constraint metadata like the code shown in the section, “Built-in constraint annotations.”

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
  xmlns=http://jboss.org/xml/ns/javax/validation/configuration
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation=http://jboss.org/xml/ns/javax/validation/configuration validation-configuration-1.0.xsd>
<constraint-mapping>META-INF/constraints.xml</constraint-mapping>
</validation-config>
```

The constraints XML file is also located in the META-INF directory and looks like the following:

```
<constraint-mappings
 xmlns=http://jboss.org/xml/ns/javax/validation/mapping
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd>
<default-package>com.my.company</default-package>
<bean class="MyConnector" ignore-annotations="true">
 <field name="serverName">
```

```
  <valid/>
  <!-- @NotNull() -->
  <constraint annotation="javax.validation.constraints.NotNull">
   <message>Value is not null</message>
  </constraint>
 </field>
 <field name="instanceCount">
  <valid/>
  <!-- @Min(1) -->
  <constraint annotation="javax.validation.constraints.Min">
    <message>Minimum possible value is 1</message>
    <element name="value">1</element>
   </constraint>
  </field>
 </bean>
<constraint-mapping>
```

The packaged RAR module, `MyResourceAdapter.rar`, looks like the following:

```
my/
  company/
    MyConnector.class
. . .
META-INF
  /validation.xml
  /constraints.xml
```

Third-party bean validation

WebSphere Application Server supports using different bean validation implementations. If a resource adapter requires a bean validation implementation different from the implementation that is provided by the product, and the RAR provides the bean validation implementation, you must package the JAR file that contains the bean validation implementation in the RAR module root directory.

The RAR module must also contain a single validation configuration descriptor (`validation.xml`), which can be packaged in the META-INF directory of the RAR module, or in the `META-INF/services` directory of the bean validation JAR file, but not both.

RAR bean validation configuration discovery

When validating RAR beans, the Application Server bootstraps the bean validation configuration, specific to the RAR, according to the bean validation descriptor supplied in the RAR META-INF directory. If the descriptor does not exist, the server bootstraps the configuration using the first validation descriptor discovered in the RAR class loading context, such as that supplied in a third-party bean validation that is packaged in the RAR. Finally, the server uses the default validation configuration provided by the product.

The server then creates a validator factory specific to the discovered bean validation configuration and uses this factory to create validator instances for validating the RAR bean instances. When you deploy a RAR that supplies a bean validation descriptor, you must take additional steps to ensure that the class loader that loads the RAR loads the bean validation descriptor and classes packaged in the RAR.

For an embedded RAR, after you have deployed the application that embeds the RAR, you must set the delegation mode of the application class loader to Parent-Last (Child-First). See the topic Configuring application class loaders for more information.

For a stand-alone RAR, you must install the RAR as an isolated resource provider. See the topic Resource Adapter settings for more information.

Troubleshooting bean validation in RAR modules:

RAR beans that fail validation are not placed into service. When constraint violations occur, applications encounter resource connectivity issues that are different according to the bean type and how the RAR is deployed. This topic explains how to understand, service, and prevent these known issues.

RAR bean constraint violations

WebSphere Application Server displays a constraint violation exception and reports all constraint violations to the system log when it validates RAR bean instances that violate one or more constraints. The cause of all constraint violation must be determined and resolved to restore full connectivity to the affected resource.

Problem determination starts with consulting the RAR provider documentation for the valid values of the configuration properties that are indicated in the violations. If the property values are invalid, you must reconfigure them according to the documentation and restart the resource adapter. If the adapter is embedded in an application, then restart the application to restart the adapter; if the adapter is stand-alone, then restart the application server.

If a valid configuration property value is indicated in a violation, then the constraint might be incorrectly specified for the bean, or the bean is incorrectly computing the property value. In these cases, the RAR vendor must correct the problem.

If the problem is caused by a faulty constraint definition (implementation), then the bean validation provider must correct the problem. In these cases, if the RAR is provided by IBM, or the RAR uses the bean validation implementation supplied by the Application Server, then contact IBM support to continue problem determination.

ResourceAdapter beans

ResourceAdapter beans are validated when the server starts a Java 2 Connector (J2C) resource adapter. When validation fails, the server rejects the ResourceAdapter instance and the resulting constraint violation exception causes the J2C resource adapter to fail. Applications cannot establish outbound connections to the resource, and the resource cannot deliver messages to applications. For an embedded adapter, the application that embeds the adapter fails to start. In-doubt transactions that involve the resource cannot be recovered.

The following example is a ResourceAdapter bean, MyConnector, at heap address 7efa7efa. Two validation constraints are violated. The constraint violation exception causes J2CResourceAdapter_1285109360562 to fail:

```
[9/29/10 10:51:24:125 CDT] 00000000 BeanValidatio E
 J2CA0238E: JavaBean com.my.company.adapter.MyConnector@7efa7efa failed Bean Validation due to one or more invalid
 configuration settings indicated in the following list of constraint violations:

 ConstraintViolationImpl{interpolatedMessage='The minimum size is 2', propertyPath=dataBaseName, rootBeanClass=class
    com.my.company.adapter.MyConnector, messageTemplate='The minimum size is 2'}
 ConstraintViolationImpl{interpolatedMessage='must be greater than or equal to 10', propertyPath=idleTimeout, rootBeanClass=class
    com.my.company.adapter.MyConnector, messageTemplate='{javax.validation.constraints.Min.message}'}
...
[9/29/10 10:51:24:468 CDT] 00000000 RALifeCycleMa E
 J2CA0128E: An Exception occurred while trying to start ResourceAdapter
  cells/IBM-46DF84D297BNode01Cell/nodes/IBM-46DF84D297BNode01/resources.xml#J2CResourceAdapter_1285109360562. The exception is:
com.ibm.ejs.j2c.metadata.ConstraintViolationException
 at com.ibm.ejs.j2c.metadata.BeanValidationHelper.validate(
 at com.ibm.ejs.j2c.RAWrapperImpl.createAndConfigureRA(
 at com.ibm.ejs.j2c.RAWrapperImpl.startRA(
 at com.ibm.ejs.j2c.RALifeCycleManagerImpl.startRA(
 at com.ibm.ejs.j2c.RALifeCycleManagerImpl.resourceProviderEvent(
. . .
```

ManagedConnectionFactory beans

ManagedConnectionFactory JavaBeans are validated during the initial Java Naming and Directory Interface (JNDI) lookup of a J2C connection factory.

When validation fails, the Application Server rejects the ManagedConnectionFactory instance and displays a naming exception to the application that performs the lookup. This exception indicates the causal constraint violation exception (javax.validation.ConstraintValidationException).

Applications cannot establish outbound connections to the resource. In-doubt transactions started over connections to the resource that were created by the connection factory cannot be recovered.

The following example is a ManagedConnectionFactory bean, MyMcf, at heap address 7dd07dd0, Two validation constraints are violated. The constraint violation exception causes the application to not obtain a connection factory that is required to create a connection to the resource, MyConnector:

```
[9/30/10 7:58:58:734 CDT] 00000023 BeanValidatio E
 J2CA0238E: JavaBean com.my.company.adapter.MyMcf@7dd07dd0 failed Bean Validation due to one or more invalid
  configuration settings indicated in the following list of constraint violations:
 ConstraintViolationImpl{interpolatedMessage='must be less than or equal to 30', propertyPath=mcfProperty2,
    rootBeanClass=class com.my.company.adapter.MyMcf, messageTemplate='{javax.validation.constraints.Max.message}'}
 ConstraintViolationImpl{interpolatedMessage='The value should be greater than 10', propertyPath=mcfProperty4,
    rootBeanClass=class com.my.company.adapter.MyMcf, messageTemplate='The value should be greater than 10'}
....
[9/30/10 7:58:58:765 CDT] 00000023 ConnectionFac E
 J2CA0009E: An exception occurred while trying to instantiate the ManagedConnectionFactory class com.my.company.adapter.MyMcf
  used by resource j2c/MyConnector : com.ibm.ejs.j2c.metadata.ConstraintViolationException
 at com.ibm.ejs.j2c.metadata.BeanValidationHelper.validate(
 at com.ibm.ejs.j2c.ServerFunction.validate(
 at com.ibm.ejs.j2c.J2CUtilityClass.createMCFEntry(
 ...
 at javax.naming.InitialContext.lookup(
 at com.my.company.app.MyEjbImpl.testJbv(
. . .
```

ActivationSpec bean violations

ActivationSpec beans are validated when the applications starts. This is when the Application Server initially activates message endpoints bound to J2C activation specifications. These activation specifications name the bean class in their configuration. When validation fails, the endpoint fails to activate and the resulting constraint violation exception causes the application hosting the endpoint to fail.

Because the J2C resource adapter that contains the activation specification is started, applications can still establish connections to the resource. The resource can deliver messages to endpoints that have successfully activated. If the activation specification is defined within an embedded resource adapter, the server stops the adapter in the course of stopping the application. Failed transactional messages delivered by previous instances of the resource adapter that contains the activation specification cannot be recovered.

The following example is an ActivationSpec bean, MyActSpec, at heap address 51625162. Two validation constraints are violated. The log shows the constraint violation exception that causes the application, my_company_app, to fail:

```
[9/29/10 10:52:05:125 CDT] 00000009 BeanValidatio E
 J2CA0238E: JavaBean com.my.company.adapter.MyActSpec@51625162 failed Bean Validation due to one or more invalid
  configuration settings indicated in the following list of constraint violations:
 ConstraintViolationImpl{interpolatedMessage='Size should be between 2 and 4', propertyPath=asProperty1,
    rootBeanClass=class com.my.company.adapter.MyActSpec, messageTemplate='Size should be between 2 and 4'}
 ConstraintViolationImpl{interpolatedMessage='Should be < 30', propertyPath=asProperty2,
    rootBeanClass=class com.my.company.adapter.MyActSpec, messageTemplate='Should be < 30'}
[9/29/10 10:52:05:171 CDT] 00000009 RAWrapperImpl E
 J2CA0089E: The method activateEndpoint on ResourceAdapter JavaBean
   cells/IBM-46DF84D297BNode01Cell/nodes/IBM-46DF84D297BNode01/resources.xml#J2CResourceAdapter_1285109389828
 failed with the following exception:
  javax.resource.ResourceException: com.ibm.ejs.j2c.metadata.ConstraintViolationException
 at com.ibm.ejs.j2c.ActivationSpecWrapperImpl.validateActivation...( at com.ibm.ejs.j2c.ActivationSpecWrapperImpl.createAndInitializ...(
 at com.ibm.ejs.j2c.ActivationSpecWrapperImpl.activateEndpoint(
...
[9/29/10 10:52:05:750 CDT] 00000009 ApplicationMg A   WSVR0217I: Stopping application: my_company_app
. . .
```

AdministeredObject beans

AdministeredObject beans are validated when the server starts a J2C resource adapter that contains the administered object in its configuration. When validation fails, the server rejects the AdministeredObject instance and the resulting constraint violation exception causes the resource adapter to fail.

The following example is an AdministeredObject beans, MyAdminObj, at heap address 3a803a80. Two validation constraints are violated. The log shows the constraint violation exception that causes resource adapter to fail:

```
[9/29/10 10:51:25:125 CDT] 00000000 BeanValidatio E
 J2CA0238E: JavaBean com.my.company.adapter.MyAdminObj@3a803a80 failed Bean Validation due to one or more invalid
  configuration settings indicated in the following list of constraint violations:
 ConstraintViolationImpl{interpolatedMessage='The value should be greater than 10', propertyPath=aoProperty4,
  rootBeanClass=class com.my.company.adapter.MyAdminObj, messageTemplate='The value should be greater than 10'}
...
[9/29/10 10:51:25:218 CDT] 00000000 AdminObjectSe A
 J2CA0017I: An exception occurred while building the serializable for JNDI deployment of jms/MyAdminObj :
 com.ibm.ejs.j2c.metadata.ConstraintViolationException
 at com.ibm.ejs.j2c.metadata.BeanValidationHelper.validate(
 at com.ibm.ejs.j2c.metadata.BeanValidationHelper.validate(
 at com.ibm.ejs.j2c.AdminObjectSerBuilderImpl._createAndValidate...(
 at com.ibm.ejs.j2c.AdminObjectSerBuilderImpl.createAndValidate...(
 at com.ibm.ejs.j2c.RALifeCycleManagerImpl.startRA(
. . .
```

JCA 1.6 support for annotations in RAR modules

The Java Connector Architecture (JCA) Version 1.6 specification adds support for Java annotations in resource archive (RAR) modules. Annotations are a means of specifying metadata, or configuration data, for a RAR module in the class files that make up the module.

Before JCA 1.6, this metadata was specified only in the deployment descriptor, but now you can specify this metadata using either a deployment descriptor or annotations. Metadata that is specified in annotations is merged into the deployment descriptor of a RAR module when it is updated, if the module is not marked metadata-complete in the deployment descriptor and if the module version is 1.6 or later.

The metadata-complete element defines whether the deployment descriptor for the resource adapter module is complete or whether the class files that are available to the module and packaged with the resource adapter should be examined for annotations that specify deployment information. If the metadata-complete is set to *true*, the application server deployment tool must ignore any annotations that specify deployment information, which might be present in the class files of the application. If metadata-complete is not specified, or is set to *false*, the deployment tool must examine the class files of the application for annotations, as specified by the JCA 1.6 Specification. If the deployment descriptor is not included, or is included but not marked metadata-complete, the deployment tool processes annotations.

Application servers must assume that metadata-complete is *true* for resource adapter modules with deployment descriptors that meet the requirements of JCA specification 1.5 and earlier. For a complete list of the supported annotations and their usage, consult the JCA specification.

The JCA Version 1.6 specification also adds support for Bean Validation constraint annotations in RAR modules. You can specify Bean Validation constraint metadata for RAR JavaBeans by decorating your classes with Bean Validation constraint annotations or by supplying XML validation descriptors. The Application Server validates the constraints of all JCA 1.6 RAR JavaBeans instances before placing them into service at run time.

Extensions to data access APIs

If a single data access API does not provide a complete solution for your applications, use WebSphere Application Server extensions to achieve interoperability between both the JCA and JDBC APIs.

Applications that draw from diverse and complex resource management configurations might require use of both the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) API and the Java Database Connectivity (JDBC) API. However, in some cases the JDBC programming model does not completely integrate with the JCA (even though full integration is a foundation of the JCA specification). These inconsistencies can limit data access options for an application that uses both APIs. WebSphere Application Server provides API extensions to resolve the compatibility issues.

For example:

Without the benefit of an extension, applications using both APIs cannot modify the properties of a shareable connection after making the connection request, if other handles exist for that connection. (If no other handles are associated with the connection, then the connection properties can be altered.) This limitation stems from an incompatibility between the connection-configuration policies of the APIs:

The Connector Architecture (JCA) specification supports relaying to the resource adapter the specific properties settings at the time you request the connection (using the getConnection() method) by passing in a `ConnectionSpec` object. The ConnectionSpec object contains the necessary connection properties used to get a connection. After you obtain a connection from this environment, your application does not need to alter the properties. The JDBC programming model, however, does not have the same interface to specify the connection properties. Instead, it gets the connection first, then sets the properties on the connection.

WebSphere Application Server provides the following extensions to fill in such gaps between the JDBC and JCA specifications:

- WSDataSource interface - this interface extends the javax.sql.DataSource class, and enables a component or an application to specify the connection properties through the WebSphere Application Server JDBCConnectionSpec class to get a connection.
  - getConnection(JDBCConnectionSpec) - this method returns a connection with the properties specified in the JDBCConnectionSpec class.
  - For more information see the `WSDataSource` API documentation topic (as listed in the API documentation index).
- JDBCConnectionSpec interface - this interface extends the `com.ibm.websphere.rsadapter.WSConnectionSpec` class, which extends the javax.resources.cci.ConnectionSpec class. The standard ConnectionSpec interface provides only the interface marker without any get() and set() methods. The WSConnectionSpec and the JDBCConnectionSpec interfaces define a set of get() and set() methods used by the WebSphere Application Server run time. This interface enables the application to specify all the essential connection properties in order to get an appropriate connection. You can create this class from the WebSphere WSRRAFactory class. For more information see the `JDBCConnection` API documentation topic (as listed in the API documentation index).
- WSRRAFactory class - this is a factory class for the WebSphere Relational Resource Adapter, which allows the user to create a JDBCConnectionSpec object or other resource adapter related object. For more information see the `WSRRAFactory` API documentation topic (as listed in the API documentation index).
- WSConnection interface - this is an interface that allows users to call WebSphere proprietary methods on SQL connections; those methods are:
  - setClientInformation(Properties props) - See the topic, Example: Setting the client information with the setClientInformation(Properties) API, for more information and examples of setting client information.
  - Properties getClientInformation() - This method returns the properties object that is set using `setClientInformation(Properties)`. Note that the properties object returned is not affected by implicit settings of client information.
  - WSSystemMonitor getSystemMonitor() - This method returns the SystemMonitor object from the backend database connection if the database supports System Monitors. The backend database will provide some connection statistics in the SystemMonitor object. The SystemMonitor object returned is wrapped in a WebSphere object (`com.ibm.websphere.rsadapter.WSSystemMonitor`) to shield applications from dependency on any database vendor code. See `com.ibm.websphere.rsadapter.WSSystemMonitor` Java documentation for more information. The following code is an example of using the `WSSystemMonitor` class:

```
import com.ibm.websphere.rsadapter.WSConnection;
...
try{
 InitialContext ctx=new InitialContext();
 // Perform a naming service lookup to get the DataSource object.
 DataSource ds=(javax.sql.DataSource]ctx.lookup("java:comp/jdbc/myDS");
} catch (Exception e) {;}
```

```
WSConnection conn=(WSConnection)ds.getConnection();
WSSystemMonitor sysMon=conn.getSystemMonitor();
if (sysMon!=null) // indicates that system monitoring is supported on the current backend database
 {
  sysMon.enable(true);
  sysMon.start(WSSystemMonitor.RESET_TIMES);
  // interact with the database
  sysMon.stop();
  // collect data from the sysMon object
 }
conn.close();
```

The WSConnection interface is part of the `plugins_root/com.ibm.ws.runtime.jar` file.

Example: Using IBM extended APIs for database connections.

Using the WSDataSource extended API, you can code your JDBC application to define connection properties through an object prior to obtaining a connection. This behavior increases the chances that the application can share a connection with another component, such as a CMP.

If your application runs with a shareable connection that might be shared with other container-managed persistence (CMP) beans within a transaction, it is recommended that you use the WebSphere Application Server extended APIs to get the connection. When you use these APIs, you cannot port your application to other application servers.

You can code with the extended API directly in your JDBC application; instead of using the DataSource interface to get a connection, use the WSDataSource interface. The following code segment illustrates WSDataSource:

```
import com.ibm.websphere.rsadapter.*;

...

// Create a JDBCConnectionSpec and set connection properties. If this connection is shared with
the CMP bean, make sure that the isolation level is the same as the isolation level that is mapped by
 the Access Intent defined on the CMP bean.

JDBCConnectionSpec connSpec = WSRRAFactory.createJDBCConnectionSpec();

connSpec.setTransactionIsolation(CONNECTION.TRANSACTION_REPEATABLE_READ);

connSpec.setCatalog("DEPT407");


//Use WSDataSource to get the connection

Connection conn = ((WSDataSource)datasource).getConnection(connSpec);
```

Example: Using IBM extended APIs to share connections between CMP beans and BMP beans.

Within an application component that accesses data through JDBC objects (such as a bean-managed persistence (BMP) bean), you can use a WebSphere extended API to define connection properties through an object prior to obtaining a connection. This behavior increases the chances that the BMP bean can share a connection with a container-managed persistence (CMP) bean.

If your BMP bean runs with a shareable connection that might be shared with other container-managed persistence (CMP) beans within a transaction, it is recommended that you use the WebSphere Application Server extended APIs to get the connection. When you use these APIs, you cannot port your application to other application servers.

In this case, use the extended API WSDataSource interface rather than the DataSource interface. To ensure that both the CMP and bean-managed persistence (BMP) beans are sharing the same physical connection, define the same access intent profile on both the CMP and BMP beans. Inside your BMP method, you can get the right isolation level from the relational resource adapter helper class.

```
package fvt.example;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.ejb.CreateException;
import javax.ejb.DuplicateKeyException;
import javax.ejb.EJBException;
import javax.ejb.ObjectNotFoundException;
import javax.sql.DataSource;

// following imports are used by the IBM extended API
import com.ibm.websphere.appprofile.accessintent.AccessIntent;
import com.ibm.websphere.appprofile.accessintent.AccessIntentService;
import com.ibm.websphere.rsadapter.JDBCConnectionSpec;
import com.ibm.websphere.rsadapter.WSCallHelper;
import com.ibm.websphere.rsadapter.WSDataSource;
import com.ibm.websphere.rsadapter.WSRRAFactory;

/**
 * Bean implementation class for Enterprise Bean: Simple
 */

public class SimpleBean implements javax.ejb.EntityBean {
 private javax.ejb.EntityContext myEntityCtx;

 // Initial context used for lookup.

 private javax.naming.InitialContext ic = null;

 // define a JDBCConnectionSpec as instance variable

 private JDBCConnectionSpec connSpec;

 // define an AccessIntentService which is used to get
 // an AccessIntent object.

 private AccessIntentService aiService;

 // AccessIntent object used to get Isolation level

 private AccessIntent intent = null;

 // Persitence table name

 private String tableName = "cmtest";

 // DataSource JNDI name

 private String dsName = "java:comp/env/jdbc/SimpleDS";

 // DataSource

 private DataSource ds = null;

 // bean instance variables.

 private int id;
 private String name;
```

```
/**
 * In setEntityContext method, you need to get the AccessIntentService
 * object in order for the subsequent methods to get the AccessIntent
 * object.
 * Other ejb methods will call the private getConnection() to get the
 * connection which has all specific connection properties
 */

public void setEntityContext(javax.ejb.EntityContext ctx) {
 myEntityCtx = ctx;

 try {
  aiService =
   (AccessIntentService) getInitialContext().lookup(
    "java:comp/websphere/AppProfile/AccessIntentService");
  ds = (DataSource) getInitialContext().lookup(dsName);
 }
 catch (javax.naming.NamingException ne) {
  throw new javax.ejb.EJBException(
   "Naming exception:"  + ne.getMessage());
 }
}

 /**
 * ejbCreate
 */

public fvt.example.SimpleKey ejbCreate(int newID)
 throws javax.ejb.CreateException, javax.ejb.EJBException {
 Connection conn = null;
 PreparedStatement ps = null;

 // Insert SQL String

 String sql = "INSERT INTO"  + tableName +  "(id, name) VALUES (?, ?)";

 id = newID;
 name = "";

 try {
                   // call the common method to get the specific connection

  conn = getConnection();
 }
 catch (java.sql.SQLException sqle) {
  throw new EJBException("SQLException caught:"  + sqle.getMessage());
 }
 catch (javax.resource.ResourceException re) {
  throw new EJBException(
   "ResourceException caught:"  + re.getMessage());
 }

 try {
  ps = conn.prepareStatement(sql);
  ps.setInt(1, id);
  ps.setString(2, name);

  if (ps.executeUpdate() != 1) {
   throw new CreateException("Failed to add a row to the DB");
  }
 }
 catch (DuplicateKeyException dke) {
  throw new javax.ejb.DuplicateKeyException(
   id + "has already existed");
 }
 catch (SQLException sqle) {
  throw new javax.ejb.CreateException(sqle.getMessage());
```

```
 }
 catch (CreateException ce) {
  throw ce;
 }
 finally {
  if (ps != null) {
   try {
    ps.close();
   }
   catch (Exception e) {
   }
  }
 }
 return new SimpleKey(id);
}

 /**
 * ejbLoad
 */

public void ejbLoad() throws javax.ejb.EJBException {

 Connection conn = null;
 PreparedStatement ps = null;
 ResultSet rs = null;

 String loadSQL = null;

 try {
                   // call the common method to get the specific connection

  conn = getConnection();
 }
 catch (java.sql.SQLException sqle) {
  throw new EJBException("SQLException caught:"  + sqle.getMessage());
 }
 catch (javax.resource.ResourceException re) {
  throw new EJBException(
   "ResourceException caught:"  + re.getMessage());
 }

 // You need to determine which select statement to be used based on the
 // AccessIntent type:
 // If READ, then uses a normal SELECT statement. Otherwise uses a
 // SELECT...FORUPDATE statement
 // If your backend is SQLServer, then you can use different syntax for
 // the FOR UPDATE clause.

 if (intent.getAccessType() == AccessIntent.ACCESS_TYPE_READ) {
  loadSQL = "SELECT * FROM"  + tableName +  "WHERE id = ?";
 }
 else {
  loadSQL = "SELECT * FROM"  + tableName +  "WHERE id = ? FOR UPDATE";
 }

 SimpleKey key = (SimpleKey) getEntityContext().getPrimaryKey();

 try {
  ps = conn.prepareStatement(loadSQL);
  ps.setInt(1, key.id);
  rs = ps.executeQuery();
  if (rs.next()) {
   id = rs.getInt(1);
   name = rs.getString(2);
  }
  else {
   throw new EJBException("Cannot load id ="  + key.id);
```

```
  }
 }
 catch (SQLException sqle) {
  throw new EJBException(sqle.getMessage());
 }
 finally {
  try {
   if (rs != null)
    rs.close();
  }
  catch (Exception e) {
  }
  try {
   if (ps != null)
    ps.close();
  }
  catch (Exception e) {
  }
  try {
   if (conn != null)
    conn.close();
  }
  catch (Exception e) {
  }
 }
}

       /**
        * This method will use the AccessIntentService to get the access intent;
        * then gets the isolation level from the DataStoreHelper
        * and sets it in the connection spec; then uses this connection
        * spec to get a connection which has the specific connection
        * properties.
        **/

private Connection getConnection()
 throws java.sql.SQLException, javax.resource.ResourceException, EJBException {

 // get current access intent object using EJB context
 intent = aiService.getAccessIntent(myEntityCtx);

 // Assume this bean only supports the pessimistic concurrency
 if (intent.getConcurrencyControl()
  != AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC) {
  throw new EJBException("Bean supports only pessimistic concurrency");
 }

 // determine correct isolation level for currently configured database
 // using DataStoreHelper
 int isoLevel =
  WSCallHelper.getDataStoreHelper(ds).getIsolationLevel(intent);
  connSpec = WSRRAFactory.createJDBCConnectionSpec();
 connSpec.setTransactionIsolation(isoLevel);

 // Get connection using connection spec
 Connection conn = ((WSDataSource) ds).getConnection(connSpec);
 return conn;
}
```

Recreating database tables from the exported table data definition language

When the WebSphere Application Server deployment tooling deploys an EJB jar file containing container-managed persistence (CMP) enterprise beans, it selects the target database and creates a corresponding *Table*.ddl file. This file contains the SQL statement necessary to generate the database table for your CMP beans.

About this task

The following steps demonstrate the process for creating tables in DB2.

Procedure

1. Extract the *Table*.ddl file from your CMP enterprise bean JAR file and save it on your database server.
   - Save the file to a temporary directory on your work station. Transfer the file to a data set on your DB2 for z/OS system.
2. Run the *Table*.ddl file.
   - Specify the data set as the input data set to SPUFI, and run the program.

Results

The database tables are created.

Container managed persistence bean associated technologies

WebSphere Application Server delivers container-managed persistence (CMP) services beyond the standards set by the Enterprise JavaBeans (EJB) specification.

According to the specification, the EJB container synchronizes the state of CMP beans with the underlying database, and manages the relationships (container-managed relationships, or CMR's) among entity beans. Thus the EJB specification relieves bean developers from writing any database-specific code; instead, they can focus on writing business logic. WebSphere Application Server offers the following additional CMP functions to increase development efficiency even more, as well as optimize the run-time performance of business logic:

Entity bean inheritance

Inheritance is a key aspect of object-oriented software development and is a capability currently missing from the EJB specification.

The use of inheritance enables a developer to define fields, relationships, and business logic in a superclass entity bean that are inherited by all subclasses. See the section *EJB inheritance* of the Rational Application Developer documentation for details on using inheritance with WebSphere Application Server and entity beans.

Access Intent Policies

Access intent policies provide Java Platform, Enterprise Edition (Java EE) application developers the mechanism by which they can indicate the intent of an application's interaction with the essential state for entity beans in order that the persistence mechanisms can make appropriate optimizations. For example, if it is known that an entity is not updated during the course of a transaction, then the persistence management is able to ease up on the concurrency control and still maintain data integrity by disallowing update operations on that bean for the duration of the transaction.

Caching data across transactions

Data caching across transactions is a configurable option set by the bean deployer that can greatly improve performance. Essentially, this is for data that changes infrequently. The option is known as *LifetimeInCache*. The data for an entity configured for lifetime in cache is stored in a cache until its specified lifetime expires. Requests on the entity during that configured lifetime use the cached data, and do not result in the execution of queries against the underlying data store. Lifetime can be expressed as time elapsed since the data was retrieved from the data store or until a specific time of day or week. The *LifetimeInCache* value can be one of the following:

Off The LifetimeInCache setting is ignored. Beans of this type are only cached in a transaction scoped cache. The cached data for this instance is not valid when the transaction is completed.

ElapsedTime

The value in the LifetimeInCache setting is added to the current time when the transaction

(in which the bean instance is retrieved) is completed. The cached data for this instance is not valid after this time. The value of the LifetimeInCache setting can add up to minutes, hours, days, and so on.

ClockTime

The value of LifetimeInCache represents a particular time of day. The value is added to the immediately preceding or following midnight to calculate a future time value, which is then treated as for Elapsed Time. Using this setting enables you to specify that all instances of this bean type have their cached data invalidated at a specific time no matter when the data were retrieved.

The use of preceding or following midnight to calculate a future time value depends on the value of LifetimeInCache. If LifetimeInCache plus preceding midnight is earlier than the current time, then the following midnight is used.

When you use the ClockTime setting, the value of LifetimeInCache must not represent more than 24 hours. If it does, the cache manager subtracts increments of 24 hours from it until a value less than or equal to 24 hours is achieved. To invalidate data at 12 midnight, you set LifetimeInCache to zero (0).

WeekTime

This setting is similar to ClockTime, except the value of LifetimeInCache is added to the preceding or following Sunday midnight (actually, 11:59 PM on Saturday plus 1 minute). In this case, the LifetimeInCache value can represent more than 24 hours, but not more than 7 days.

See the *LifetimeInCache* help sections of the assembly tool for more details.

Note:

Because the data used by an entity bean can be loaded by previous transactions, if you configure the bean as LifeTimeInCache, the isolation level and update lock (access intent policies) for the bean are lost for the current transaction. This can cause data integrity problems if your application has logic to calculate information from read-only data, and then save the result in another bean. This makes it important to perform read-read consistency checking to ensure the data get locked properly if loading the data from in-memory cache; otherwise, data is updated to the database without knowing the underlining data is changed, causing previous changes to be lost. For more information, see the topic Configuring read-read consistency checking with an assembly tool.

Read-only entity beans

Declaring entity beans as read-only potentially increases the performance enhancement offered by caching. Both features operate on the same principle: to minimize the overhead incurred by frequent reloading of entity beans from data in persistent storage. When you designate entity beans as read-only, you can specify the reload requirements and frequency, according to the needs of your application.

To use this function, you declare the bean type as read-only by selecting a particular set of bean caching options, through a selection list within the assembly tooling. See Configuring read-read consistency checking with an assembly tool for details.

Container-managed persistence restrictions and exceptions:

Some external software that directly impact your applications can limit container-managed persistence (CMP) features. However, you can work around these limitations.

In each case, only very specific behaviors of the software place restrictions on your CMP beans. The following tips help you prevent these behaviors.

CMP deployment and Sybase IMAGE type restriction

When deploying enterprise beans with container managed persistence (CMP) types that are non-primitive and do not have a natural JDBC mapping, the deployment tool maps the CMP type to a binary type in the database, where it is stored as a serialized instance. For Sybase, the tool uses the JDBC type *LONG VARBINARY*. The Sybase driver maps *LONG VARBINARY* to the native type *IMAGE*.

Although the type *VARBINARY* has fewer restrictions than *IMAGE* in Sybase, you cannot use it because it is limited to a size of 255 bytes, which is too small for typical serialized Java objects.

The specific restrictions on the IMAGE type are:

- You cannot use the *IMAGE* type in the *WHERE* clause of an SQL query. You can encounter this restriction whenever an enterprise bean contains an EJB-QL query that has a CMP type in the *WHERE* clause, which maps to the *IMAGE* type in the Sybase relational database.
- You cannot use *IMAGE* type in select queries marked *DISTINCT*. This situation arises in these user scenarios:
  - When the *DISTINCT* key word is specified in an EJB-QL select query having a Java type mapping to IMAGE.
  - When Enterprise beans have finder and ejbSelect() methods returning java.util. Set and have CMP types mapping to *IMAGE*.

To work around this restriction, edit the EJB mappings in the Rational Application Developer toolset and do either of the following:

- If you are **sure** that the serialized instance of the CMP type is **never** larger than 255 bytes, you can change the CMP type mapping from *IMAGE* or *LONG VARBINARY* to *VARBINARY*.
- Map the CMP type to multiple RDB fields through a composer. For example, if the CMP type is a Java object X with an int field and a string field, then map X to two RDB fields *INTEGER* and *VARCHAR*, using a composer. Refer to the Rational Application Developer documentation for more information about using composers.

A ClassCastException exception occurs when running CMP 1.1 beans

If you created your Enterprise JavaBeans (EJB) application using Rational Application Developer or WebSphere Studio Application Developer Integration Edition, Version 4.0.x , and the application contains container managed persistence (CMP) 1.1 beans with associations (relationships), you might receive a *java.lang.ClassCastException* exception when you run your application on WebSphere Application Server.

Note: Business processes modeled with WebSphere Studio Application Developer Integration Edition Version 5.0 or earlier are deprecated.

The cast operation generated by Rational Application Developer or WebSphere Studio Application Developer Integration Edition, Version 4.0.x, does not use the *javax.rmi.PortableRemoteObject.narrow(...)* object to convert the remote object to the remote interface of CMP beans in the *XToYLink.java* (or *YToXLink.java*) class where X and Y are CMP 1.1 beans.

Recommended response:

1. Locate the following methods in all link classes, for example, XToYLink.java and YToXLink.java where X and Y are CMP 1.1 beans:

   ```
   public void secondaryAddElementCounterLinkOf(javax.ejb.EJBObject anEJB)
   public void secondaryRemoveElementCounterLinkOf(javax.ejb.EJBObject anEJB)
   public void secondarySetCounterLinkOf(javax.ejb.EJBObject anEJB)
   ```

2. Add the javax.rmi.PortableRemoteObject.narrow(...) object to convert the remote object to the remote interface of CMP beans.

For example, change the following original method:

```
public void secondaryAddElementCounterLinkOf(javax.ejb.EJBObject anEJB) throws java.rmi.RemoteException {
        if (anEJB != null)
               ((X) anEJB).secondaryAddY((Y) getEntityContext().getEJBObject());
```

to:

```
public void secondaryAddElementCounterLinkOf(javax.ejb.EJBObject anEJB) throws java.rmi.RemoteException {
        if (anEJB != null)
             ((X) anEJB).secondaryAddY((Y)
javax.rmi.PortableRemoteObject.narrow(getEntityContext().getEJBObject(), Y.class));
```

Application performance and entity bean behavior:

WebSphere Application Server allows you to override two behaviors that are required by the Enterprise JavaBeans (EJB) specification, because your application might benefit from handling these aspects of bean data management in a slightly different manner.

Application-managed persistent store synchronization for findBy methods

Sections 10.5.3 and 12.1.4.2 of the EJB 2.0 and 2.1 specifications require that prior to running a query as part of any findBy method (except for findByPrimaryKey), the EJB container writes out to persistent storage the state of any entity beans of the type that are enlisted in the current transaction. Stated another way, the container performs the following actions:

1. Creates a list of beans that are both enlisted in the current transaction and are of the same type that the findby method is returning
2. Stores the state of these enterprise beans to persistent storage before running the query

If the state of an EJB instance is not altered in the current transaction, the store operation is skipped for that instance. This practice ensures that the query is performed on the most current state of all the persistent data, reducing the chance of data integrity issues.

However, there are scenarios where it is inefficient and wasteful for the EJB container to automatically perform this action on every findBy method. Examples of this would be where the application itself ensures that the most current data is used on findBy queries, or where the application can tolerate some non-current data as part of the query results.

WebSphere Application Server allows you to initiate the synchronization process under application control, and to disable the container-managed synchronization for specific EJB types within your application. Careful use of these functions can improve the performance of your application without sacrificing data integrity. Refer to the topic Manipulating the synchronization of entity beans and datastores.

Avoiding ejbStore invocations on non-modified entity bean instances

The EJB specification requires that the EJB container invoke the user-provided ejbStore method on all entity beans within a transaction when that transaction is committed. For container-managed persistence (CMP) beans (as opposed to bean-managed persistence beans) this operation is usually unnecessary, because this method on CMP beans is often empty. Even in cases where the method is not empty, the application might only require the method to be called if the bean's persistent state is modified during the current transaction.

WebSphere Application Server provides a mechanism for you to indicate if you want this behavior for specific EJB types within the application. See the topic Avoiding ejbStore invocations on non-modified EntityBean instances.

Manipulating synchronization of entity beans and datastores

You can indicate that a particular Enterprise JavaBeans (EJB) type should not synchronize its state to persistent storage prior to each findBy invocation by using environment variables or a marker interface.

About this task

There are two options available for indicating that a particular EJB type should not synchronize its state to persistent storage prior to each findBy invocation:

- Set an EJB environment variable within the bean's deployment descriptor
- Have the bean implementation class implement a marker interface. This second technique is especially useful if you have a number of bean implementations that all extend a single root class; in this case you can have the root class implement the marker interface, causing all beans that extend this class to inherit the behavior as well.

Procedure

- **To use the EJB environment variable technique**, edit the EJB deployment descriptor using any standard Java Platform, Enterprise Edition (Java EE) development tool. For information on your tool options, see the topic, Assembly tools.
  1. Start the tool.
  2. Select the EJB deployment descriptor of the bean with which you want to work.
  3. Create an EJB environment variable with the name **com/ibm/websphere/ejbcontainer/disableFlushBeforeFind**.
  4. Set the type of this variable to **java.lang.Boolean**.
  5. Set the value to True to prevent the pre-find synchronization, or False to enable the default behavior.
  6. Save your changes.
- **To use a marker interface**, code your bean implementation class to implement the **com.ibm.websphere.ejbcontainer.disableFlushBeforeFind** interface. The bean implementation class need not directly implement the interface; any parent class can implement the interface. See the **com.ibm.websphere.ejbcontainer** package in the **Reference** > **Developer** > **API documentation** section of the information center.

  Ensuring data integrity for queries performed during a transaction

  If you choose to disable the automatic pre-find synchronization for certain bean types, it is very important that your application use other means to ensure that queries performed during the transaction are not performed on data that might no longer be valid. You can use the flushCache method on the com.ibm.websphere.ejbcontainer.EJBContextExtension class (an extension of javax.ejb.EJBContext) to perform a manual synchronization to persistent storage at times that are defined by the application. For more information on EJBContextExtension and its related classes SessionContextExtension, EntityContextExtension and MessageDrivenContextExtension, see the **com.ibm.websphere.ejbcontainer** package in the **Reference** > **Developer** > **API documentation** section of the information center.

Avoiding ejbStore invocations on non-modified EntityBean instances

You can configure your EntityBean instances to bypass an invocation of the ejbStore method if they have not been modified during the current transaction.

About this task

There are two options available for indicating that a particular Enterprise JavaBeans (EJB) type should only have its ejbStore method invoked if the bean has been modified during the current transaction:

- Set an EJB environment variable within the bean's deployment descriptor
- Have the bean implementation class implement a marker interface. This second technique is especially useful if you have a number of bean implementations that all extend a single root class; in this case you may have the root class implement the marker interface, causing all beans that extend this class to inherit the behavior as well.

Procedure

- **To use the EJB environment variable technique**, edit the EJB deployment descriptor using any standard Java Platform, Enterprise Edition (Java EE) development tool. Use the following steps as a guide. For information on your tool options, see the topic, Assembly tools.
  1. Start the tool.
  2. Select the EJB deployment descriptor of the bean you want to work with.
  3. Create an EJB environment variable with the name **com/ibm/websphere/ejbcontainer/ disableEJBStoreForNonDirtyBeans**.
  4. Set the type of this variable to **java.lang.Boolean.**
  5. Set the value to True to avoid the ejbStore invocation, or False to enable the default behavior.
  6. Save your changes.
- **To use a marker interface**, code your bean implementation class to implement the **com.ibm.websphere.ejbcontainer.DisableEJBStoreForNonDirtyBeans** interface. The bean implementation class need not directly implement the interface; any parent class can implement the interface. See the **com.ibm.websphere.ejbcontainer** package in the **Reference** > **Developer** > **API documentation** section of the information center.

Resource reference benefits

WebSphere Application Server requires your code to reference application server resources (such as data sources or J2C connection factories) through logical names, rather than access the resources directly in the Java Naming and Directory Interface (JNDI) name space. These logical names are called *resource references*.

Application Server requires use of resource references for the following reasons:

- If application code looks up a data source directly in the JNDI naming space, every connection that is maintained by that data source inherits the properties that are defined in the application. Consequently, you create the potential for numerous exceptions if you configure the data source to maintain shared connections among multiple applications. For example, an application that requires a different connection configuration might attempt to access that particular data source, resulting in application failure.
- It relieves the programmer from having to know the name of the actual data source or connection factory at the target application server.
- You can set the default isolation level for a data source through resource references. With no resource reference you get the default for the JDBC driver you use.

The following example of using a resource reference invokes a data source by creating a place holder for it through the lookup method. Using the logical name *jdbc/Section*, the code stores the place holder in the JNDI subcontext *java:comp/env/*; hence *jdbc/Section* becomes a resource reference. (The subcontext *java:comp/env/* is the name space that WebSphere Application Server provides exclusively for object references within application code.)

```
javax.sql.DataSource specificDataSource  =
   (javax.sql.DataSource) (new InitialContext()).lookup("java:comp/env/jdbc/Section");
//The method InitialContext()).lookup creates the logical name, or resource reference, jdbc/Section.
```

Generally, an actual data source is configured later as an administrative task.

The logical name *jdbc/Section* is officially declared as a resource reference in the application deployment descriptor. You can then associate the resource reference with the JNDI name of the actual data source in several ways:

- If you know the data source JNDI name at the point of application assembly, specify the name on the resource references Bindings page.
- Specify the data source JNDI name during application deployment.

- Map the resource reference to the data source JNDI name when you configure the application after deployment.

This act of association is called *binding* the resource reference to the data source.

See the article, Application bindings, for information on all types of required resource bindings.

Requirements for setting data access isolation levels:

This article discusses the criteria and effects of setting isolation levels for data access components that comprise Enterprise JavaBeans (EJB) 2.x and later modules.

In an EJB 1.1 module, you can set the isolation level at the method level or bean level. This capability also applies to container-managed persistence (CMP) 1.1 beans that you assemble into *EJB 2.x modules*. WebSphere Application Server permits the deployment descriptor of a CMP bean to declare the version level of 1.1, regardless of the overall module version.

However, the ability to set isolation level at the method or bean level does **not** apply to other enterprise beans within an EJB 2.x module, including *CMP 2.x beans*. WebSphere Application Server Version 5.0 removed this capability from EJB 2.0 modules to deliver an architecture that ultimately provides more efficient connection use.

Consequently, later versions of the product enforce the following restrictions on declaring isolation level for CMP 2.x beans—as well as session beans, message-driven beans, and bean managed persistence (BMP) beans that you assemble into EJB 2.x modules:

- You cannot specify isolation level on the EJB method level or bean level.
- If you configure a JDBC application, a bean-managed persistence (BMP) bean, or a servlet to participate in global transactions, any connection that is shared cannot accept a user-specified isolation level. WebSphere Application Server can only set a user-specified isolation level on a connection that is not shared within a global transaction. *Generally, you want to refrain from specifying isolation levels on shareable connections.*

The configuration for the isolation level is determined by the type of bean that is used by the component:

Isolation level on connections used by 2.x CMP beans

In a EJB 2.x module, when a CMP 2.x bean uses a new data source to access a backend database, the isolation level is determined by the WebSphere Application Server run time, based on the type of access intent assigned to the bean or the calling method. Other non-CMP connection users can access this same data source and also use the access intent and application profile support to manage their concurrency control.

Connections used by other 2.x enterprise beans and other non-CMP components

For all other JDBC connection instances (connections other than those used by CMP beans), you can specify an isolation level on the data source resource reference. For shareable connections that run in global transactions, this method is the only way to set the *isolationLevel* for connections. Trying to directly set the isolation level through the *setTransactionIsolation()* method on a shareable connection that runs in a global transaction is not allowed. To use a different isolation level on connections, you must provide a different resource reference. Set these defaults through your assembly tool.

Each resource reference associates with one isolation level. When your application uses this resource reference Java Naming and Directory Interface (JNDI) name to look up a data source, every connection returned from this data source using this resource reference has the same isolation level.

Components needing to use shareable connections with multiple isolation levels can create multiple resource references, giving them different JNDI names, and have their code look up the

appropriate data source for the isolation level they need. In this way, you use separate connections with the different isolation levels enabled on them.

It is possible to map these multiple resource references to the same configured data source. The connections still come from the same underlying pool, however; the connection manager does not allow sharing of connections requested by resource references with different isolation levels. Consider the following scenario:

- A data source is bound to two resource references: *jdbc/RRResRef* and *jdbc/RCResRef*.
- RRResRef has the *RepeatableRead* isolation level defined. RCResRef has the *ReadCommitted* isolation level defined.

If your application wants to update the tables or a BMP bean updates some attributes, it can use the *jdbc/RRResRef* JNDI name to look up the data source instance. All connections returned from the data source instance have a RepeatableRead isolation level. If the application wants to perform a query for read only, then it is better to use the *jdbc/RCResRef* JNDI name to look up the data source.

The product does not require you to set the isolation level on a data source resource reference for a non-CMP application module. If you do not specify isolation level on the resource reference, or if you specify TRANSACTION_NONE, the WebSphere Application Server run time uses a default isolation level for the data source. Application Server uses a default setting based on the JDBC driver.

For most drivers, WebSphere Application Server uses an isolation level default of TRANSACTION_REPEATABLE_READ. For Oracle drivers, however, Application Server uses an isolation level of TRANSACTION_READ_COMMITTED. Use the following table for quick reference:

Database:	**Default isolation level:**
DB2	RR
Oracle	RC
Sybase	RR
Informix®	RR
Apache Derby	RR
SQL Server	RR

Note: These same default isolation levels are used in cases of direct JNDI lookups of a data source.

- RR = JDBC Repeatable read (TRANSACTION_REPEATABLE_READ)
- RC = JDBC Read committed (TRANSACTION_READ_COMMITTED)

To customize the default isolation level, you can use the webSphereDefaultIsolationLevel custom property for the data source. In most cases you should define the isolation level in the deployment descriptor when you package the EAR file, but in certain situations you might need to customize the default isolation level. This property will have no effect if any of the previous options are used, and this custom property is provided for those situations in which there is no other means of setting the isolation level.

Use the following values for webSphereDefaultIsolationLevel custom property:

Possible values	**JDBC isolation level**	**DB2 isolation level**
8	TRANSACTION_SERIALIZABLE	Repeatable Read (RR)
4 (default)	TRANSACTION_REPEATABLE_READ	Read Stability (RS)
2	TRANSACTION_READ_COMMITTED	Cursor Stability (CS)

Possible values	JDBC isolation level	DB2 isolation level
1	TRANSACTION_READ_UNCOMMITTED	Uncommitted Read (UR)

To define this custom property for a data source:

1. Click **Resources** > **JDBC provider** > ***JDBC_provider***.
2. Click **Data sources** in the Additional Properties section.
3. Click the name of the data source.
4. Click **Custom properties**.
5. Create the webSphereDefaultIsolationLevel custom property.
   a. Click **New**.
   b. Enter webSphereDefaultIsolationLevel for the name field.
   c. Enter one of the possible values in the value field.

Application Server sets the isolation level by prioritizing the available settings. Application Server will set the isolation level based on the values for the following, in this order:

1. Resource reference isolation level
2. Isolation level that is specified by the access intent policy
3. Custom property that configures an isolation level
4. Application Server's default setting.

Data source lookups for enterprise beans and web modules:

During either application assembly or deployment, you must bind the resource reference to the Java Naming and Directory Interface (JNDI) name of the actual resource in the runtime environment. You can take this action in the assembly tool or as one of the steps during installation of the application enterprise archive (EAR) file.

Bean-managed persistence bean: When developing your bean-managed persistence (BMP) bean you generally lack knowledge about the name of the data source on the target application server. In your code, do not look up the data source directly. Instead, you look up the resource reference from the *java:comp/env namespace* file. Let us assume that you look up the resource reference named *ref/ds*, for example:

```
javax.sql.DataSource dSource = (javax.sql.DataSource)((new InitialContext()).lookup("java:comp/env/ref/ds"));
```

In the assembly tool, you specify the name **ref/ds** in the Resource Reference page on the General Tab. If you know the name of the data source you can specify it in this Resource References page on the Bindings Tab. Note that if you do not specify it here , you must provide this JNDI name when you install the application EAR file.

Container-managed persistence bean: The data source binding process for the container-managed persistence (CMP) bean is the same process that you perform for bean-managed persistence (BMP) beans. Use the data source JNDI name as a WebSphere binding property for each bean during application assembly.

Servlets and JavaServer Pages Files: In a servlet application, you look up the data source exactly as you look it up in the BMP bean case.

Direct and indirect JNDI lookup methods for data sources:

You can use a direct or indirect method for the Java Naming and Directory Interface (JNDI) name (such as jdbc/DataSource) to look up a data source.

Direct

When you use a JNDI name such as jdbc/myDatasource, the application server assigns default values to the resource reference data. An informational message resembling the following is logged to document the default values:

```
[10/5/07 11:40:38:468 CDT] 0000002e ConnectionFac W
 J2CA0294W: Direct JNDI lookup of resource jdbc/myDatasource.
   The following default values are used:

[Resource-ref CMConfigData key items]
res-auth:                 1 (APPLICATION)
res-isolation-level:      0 (TRANSACTION_NONE)
res-sharing-scope:        true (SHAREABLE)
loginConfigurationName:   null
loginConfigProperties:    null

[Resource-ref non-key items]
isCMP1_x:                 false (not CMP1.x)
isJMS:                    false (not JMS)
commitPriority            0
Java EE Name:                not set
Resource ref name:        not set
isCMP:                    false (not set)
```

The first of these attributes, *res-auth*, dictates what type of authentication is done. This default setting says that the component-managed authentication alias is used if you do not specify an activation specification or you do not specify the username and password on the getConnection call. It says that the container-managed alias is not used.

The second of these settings, *res-isolation-level*, says that the isolation level is set to the "default" settings. For an enterprise bean, you can set this in the Enterprise JavaBeans (EJB) bean itself. For a servlet getting a connection, this results in the isolation level being Repeatable_Read. This is a fairly restrictive isolation level. This can lead to lowered performance, because application requests will lock more rows than with a less restrictive isolation level.

Finally, the *res-sharing-scope* is set to **Shareable**, meaning a Shareable connection is used. For some applications, a Shareable connection is fine. For others, in particular those servlets that get multiple connections within a single service() method, it is not.

To avoid any surprises that might accompany these settings, you should change your application to use an indirect JNDI name instead of the direct JNDI name, and you should create a resource reference.

Indirect

To use values that are different from the defaults, use an assembly tool to define your resource reference. The resource reference can also be created in the EJB Deployment Descriptor (ejb-jar.xml), Web Deployment Descriptor (web.xml), or Application Client Deployment Descriptor (application-client.xml) editors using an assembly tool. After you define the resource reference, you can do an indirect JNDI lookup (using the java:comp/env context). Then the values for the resource reference properties that are defined in the resource reference are used and the J2CA0122I message no longer appears. Read the topic on creating a resource reference for more information.

Access intent service:

The access intent service enables developers to precisely tune the management of application persistence.

Access intent enables developers to configure applications so that the Enterprise JavaBeans (EJB) container and its agents can make performance optimizations for entity bean access. Entity beans and entity bean methods are configured with access intent policies. A policy is acted upon by either the

combination of the WebSphere EJB container and Persistence Manager (for container-managed persistence (CMP) entities) or by bean-managed persistence (BMP) entities directly. Note that access intent policies apply to entity beans only.

Predefined access intent policies

Seven predefined access intent policies are available. The policies are composed of different attributes. The *access type* is of primary interest and controls the isolation level, lock type, and duration of locks obtained when bean data is read from the database.

A pessimistic access type indicates to hold locks for the duration of the transaction under which the data loads. An optimistic type indicates to drop locks immediately after the data is read from the backend. A *read* type indicates that the run time must not allow updates to the data; any attempt to do so on data read under a *read* type results in an exception. *Update* types permit you to change data.

Though a pessimistic update policy is designed to hold update locks on data records, it does not block threads with other policies that try to access the same data records. When two threads that run pessimistic update policies access a given record, they serialize (but not block) other threads that run pessimistic read or optimistic policies and try to access the same record.

The seven access intent policies and their attribute definitions follow:

wsPessimisticUpdate
- Access type = Pessimistic update
- Collection scope = Transaction
- Collection increment = 1
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsOptimisticUpdate
- Access type = Optimistic update
- Collection scope = Transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsOptimisticRead
- Access type = Optimistic read
- Collection scope = Transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticRead
- Access type = Pessimistic read
- Collection scope = Transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticUpdate-Exclusive
- Access type = Pessimistic update
- Exclusive = true
- Collection scope = Transaction
- Collection increment = 1
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticUpdate-NoCollision
- Access type = Pessimistic update
- No collision = true
- Collection scope = Transaction

- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

wsPessimisticUpdateWeakestLockAtLoad

- ***default policy**
- Access type = Pessimistic Update
- Promote = true
- Collection scope = transaction
- Collection increment = 25
- Resource manager prefetch increment = 0
- Read ahead hint = null

To support connection sharing, you must ensure that all data loaded in the same transaction is under the same isolation level. Verify that all participating methods that drive loads are configured with either a pessimistic access type or an optimistic access type.

Access intent -- isolation levels and update locks:

WebSphere Application Server access intent policies provide a consistent way of defining the isolation level for CMP bean data across the different relational databases in your environment.

Within a deployed application, the combination of an access intent policy *concurrency definition* and *access type* signifies the isolation level value that Application Server sets on a database connection. See the articles, Concurrency control, and Access intent and isolation, for more information on concurrency and access type. This combination of properties also signifies the update lock flag that Application Server passes to the database through a JDBC prepared statement.

Databases do not provide as many isolation level definitions as WebSphere Application Server. Databases define an isolation level as one of only three types. Furthermore, only one parameter indicates the type of isolation level that the databases set on incoming connections. Each of the three types can be represented by a *different* parameter value, as determined by each database vendor. For example, one database might define an isolation level as RR (JDBC Repeatable read), whereas a different database might define the same isolation level as RC (JDBC Read committed).

Because of this inconsistency, WebSphere Application Server does not map access intent policies to the parameter values. Instead, Application Server maps access intent policies to the types of isolation level that are common across all database vendors.

Table 7. Access intent policies relationship to database isolation levels and update lock settings. The following matrix shows how access intent policies correspond to different database isolation levels and update lock settings.

Access Intent profile	Isolation level						Update lock implementation
	DB2	Oracle*	SyBase	Informix	Apache Derby	SQL Server	
wsPessimisticUpdate-Weakest LockAtLoad (Default policy)	RR	RC	RR	RR	RR	RR	No (*Oracle, Yes)
wsPessimisticUpdate	RR	RC	RR	RR	RR	RR	Yes
wsPessimisticRead	RR	RC	RR	RR	RR	RR	No
wsOptimisticUpdate	RC	RC	RC	RC	RC	RC	No
wsOptimisticRead	RC	RC	RC	RC	RC	RC	No
wsPessimisticUpdate No-Collisions	RC	RC	RC	RC	RC	RC	No
wsPessimisticUpdate-Exclusive	S	S	S	S	S	S	Yes

- RC = JDBC Read Committed
- RR = JDBC Repeatable Read
- S = JDBC Serializable

- * Oracle does not support JDBC Repeatable Read (RR). Therefore, wsPessimisticUpdate-weakestLockAtLoad and wsPessimisticUpdate behave the same way on Oracle as do wsPessismisticRead and wsOptimisticRead. Because of an Oracle restriction, the OracleXADataSource JDBC class cannot run with an S transaction isolation level. Therefore, you cannot use this class to run an application containing enterprise beans with access intent policies that are configured to cause the bean to load with S isolation.
- Setting access intent policies per EJB method support is deprecated for Version 6.0. It is recommended that you set access intent only for the entire bean.

New for MS SQL Server 2005: MS SQL Server 2005 offers a new option for the Read Committed isolation level and a new option for the Serializable isolation level:

- Read Committed with Snapshots
- Transaction Snapshot (for Serializable)

Both options use optimistic locking. To use Read Committed with Snapshots instead of Read Committed, enable the READ_COMMITTED_SNAPSHOT setting for the database according to the MS SQL Server 2005 documentation. To use Transaction Snapshot instead of Serializable, configure the custom data source property, snapshotSerializable, to "true" and enable the ALLOW_SNAPSHOT_ISOLATION setting for the database according to the MS SQL Server 2005 documentation.

Structured Query Language (SQL) keywords and restrictions

Table 8. SQL keywords and restrictions. The following table shows which SQL keywords are used during update intent locking, as well as any restrictions imposed on the SQL.

Database	SQL syntax used for locking update	join restrictions	order by restrictions	subselect restrictions	aggregation restrictions
DB2	FOR UPDATE OF	not allowed	not allowed	not allowed	not allowed
DB2 UDB for iSeries® (V5R3 and earlier)	FOR UPDATE OF	not allowed	allowed with limitations*	allowed with limitations*	not allowed
DB2 UDB for iSeries (V5R4 and later)	WITH RS/RR USE AND KEEP EXCLUSIVE LOCKS	not allowed	allowed with limitations*	allowed with limitations*	not allowed
DB2 on z/OS V8.x	WITH RS/RR USE AND KEEP UPDATE LOCKS	none	none	none	none
DB2 UDB workstation V8.2	WITH RS/RR USE AND KEEP UPDATE LOCKS	none	none	none	none
Oracle	FOR UPDATE	none	none	none	none
Apache Derby	FOR UPDATE OF	not allowed	not allowed	not allowed	not allowed
Informix	FOR UPDATE	not allowed	not allowed	not allowed	not allowed
Sybase	FOR UPDATE	not allowed	not allowed	not allowed	not allowed
Sqlserver	UPDLOCK	not allowed	not allowed	not allowed	not allowed

Note: For details on the limitations for these permitted SQL restrictions, refer to the DB2 Universal Database™ for iSeries SQL Reference. You can find this document in the iSeries Information Center, Version 5 Release 4. In the Contents navigation area, click **Database** > **Reference** > **SQL Reference**.

Custom finder SQL dynamic enhancement:

To ensure data integrity for applications using custom finders defined on Enterprise JavaBeans (EJB) version 1.1 home interfaces, WebSphere Application Server Version 6.x uses custom finder Structured Query Language (SQL) dynamic enhancement to maintain correct SQL locking semantics.

WebSphere Application Server uses SQL clauses applied to the custom finder SQL statements for those custom finders defined with the *Update* attribute and certain method-level isolation level settings. These dynamic enhancements are applied only if the backend data store supports these clauses.

This support takes affect at run time when the run time attempts to execute container-managed persistence (CMP) persistence operations associated with the custom finders. To ensure that the SQL dynamic enhancements occur correctly for custom finders defined on an EJB version 1.1 home interface accessing a backend data store that requires the special SQL locking clauses, WebSphere Application Server provides new Java Virtual Machine (JVM) and bean (module) properties. These properties enable you to indicate which custom finders should be enhanced, provided the backend store supports the SQL clauses. For more information about these properties, Custom finder SQL dynamic enhancement properties.

There are several important items to consider when using this functionality:

- This support **only** applies to EJB version 1.1 CMP Custom Finder methods
- Option A CMP beans and CMP beans involved in an inheritance relationship are not supported
- Applications using this capability in WebSphere Application Server for z/OS Version 4.x continue to function, but you must address some compatibility issues:
  - The default behavior of WebSphere Application Server Version 5.x and above is the opposite of the Version 4.x product, that is, the default for 5.x and above is **not** to enhance custom finder SQL statements unless directed to by specific settings. If your WebSphere Application Server for z/OS installation relies on the automatic dynamic enhancement of all custom finders in all applications installed, you must set the *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* indicator to **all**.
  - If an application contains a bean which has the *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* indicator set into its env-var settings, that indicator continues to be used, provided the dynamic SQL enhancement features of the product at Version 5.x and above are enabled. For more information, see the topic Custom finder SQL dynamic enhancement properties.

Custom finder SQL dynamic enhancement properties:

Use this page to modify custom finder SQL dynamic enhancement properties settings.

To ensure that the Structured Query Language (SQL) dynamic enhancements occur correctly for custom finders defined on an EJB 1.1 Home interface that uses a backend data store that requires the special SQL locking clauses, the following Java virtual machine (JVM) and bean (module) properties are provided. These properties enable you to indicate which custom finders to enhance, assuming the backend data store supports the SQL clauses.

For z/OS, to view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* **> Control** (to define the property in the Control) or **Servant** (to define the property in the Servant) **> Java and process management > Process definition > Java virtual machine > Custom properties**.

com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent:

Used to indicate which enterprise beans should have custom finder SQL dynamic enhancement enabled at runtime.

This property takes effect at the server level. Any EJB 1.1 home interface-defined custom finder (prefix named *find*) that has *Update* as an access intent is a candidate for custom finder SQL dynamic enhancement based on its specified isolation level. If the backend data store requires special SQL semantics, they are applied. The particular SQL used varies according to the isolation level you choose for beans in the application, as well the backend data base being used. If set to **all**, custom finder SQL

dynamic enhancement is enabled for all custom finders defined in any beans that are installed into the container. If set to **J2EENAME[:J2EENAME]**, where *J2EENAME* is a fully qualified package or bean name, custom finder SQL dynamic enhancement is enabled for only the custom finders defined in the beans that are installed into the container and represented by the bean names denoted.

Information	Value
Data type	String
Range	Valid values are **all** or **J2EENAME[:J2EENAME]**
Default	Enhancement behavior not active

Note: Some of your applications might use custom finders that have been manually coded and already contain the SQL locking clauses, or keywords *ORDER BY* and *DISTINCT* on the *SELECT* operation. In these instances, if the run time attempts SQL dynamic enhancement, the possibility exists of introducing malformed SQL statements to the underlying backend data store. If an application contains these custom finders, then you must be careful when specifying the value for the JVM property *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent*. A value of **all** causes custom finder SQL dynamic enhancement to occur for every custom finder method defined with an access intent of *Update* found in all beans that are installed in the application server, thus introducing malformed SQL for that subset of custom finders.

To prevent this from happening, ***do not*** set the server-wide setting to **all**. Instead, use the bean method level property, *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel* to indicate on a per bean basis only those custom finder methods that should have the custom finder SQL dynamic enhancement executed on them at run time.

com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel:

Used to indicate custom finder SQL dynamic enhancement be enabled at the method level on a particular bean.

When a bean is defined with this property set to a list of one or more custom finder methods, any custom finder (prefix named *find*) defined on the home interface that has a matching method name and parameter signature has SQL locking semantics applied at run time. This occurs only if the custom finder method has an access intent of *Update* specified and the backend data store supports the SQL clauses. The particular SQL used varies according to the isolation level chosen for the application as well as the backend data store being used.

Information	Value
Data type	String
Range	Valid value is a string of this form: **method1(parm1,parm2,..parmn):method2(parm1,parm2,..parmn):methodn(...)**

com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent:

Used by WebSphere Application Server for z/OS Version 4.x users to indicate that the SQL enhancement capability should **not be** applied to applications installed in the WebSphere Application Server for z/OS product.

The default behavior of the WebSphere Application Server for z/OS Version 4.x product is to perform the dynamic SQL enhancements. For those z/OS users choosing not to participate in dynamic SQL enhancement of custom finders in the Version 4.x product, this attribute is used to make this indication at both the bean and the server level.

At the bean level, a name/value pair consisting of this attribute name and a value of **true** disables the SQL enhancement of any custom finder defined on the given bean's home interface.

At the server level, an entry into the WebSphere Application Server for z/OS server property file with a value of **true** disables the SQL enhancement of all beans installed in the given server.

This custom finder enhancement attribute continues to be supported by the runtime at the bean level in the product. Its use as a server wide indicator has been deprecated by the fact that the default behavior of earlier versions is to ***not*** dynamically enhance custom finder SQL.

Note: If your WebSphere Application Server for z/OS installation relies on the automatic dynamic enhancement of all custom finders in all applications installed, you should set the *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* indicator to **all**. If an application contains a bean that has the *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* indicator set into its *env-var* settings, that indicator continues to be used, provided the dynamic SQL enhancement features of the product are enabled as described above.

Information	**Value**
Data type	String
Range	Valid values are **true** and **false**

Some notes about precedence:

- The *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent.methodLevel* attribute overrides any server-wide or bean level attribute setting
- Any bean listed through a *J2EE Name* in the *com.ibm.websphere.ejbcontainer.customfinder.honorAccessIntent* indicator causes dynamic enhancement to occur for custom finders defined for that bean, even if the default behavior is in effect for the server in question.
- The *com.ibm.websphere.persistence.bean.managed.custom.finder.access.intent* attribute disables a particular bean's use of this feature if the server-wide setting or bean setting is enabled and no method level settings are specified.

Accessing data using Java EE Connector Architecture connectors

To access data from a Java EE Connector Architecture (JCA) compliant application in WebSphere Application Server, you configure and use resource adapters and connection factories.

About this task

An application component uses a connection factory to access a connection instance, which the component then uses to connect to the underlying enterprise information system (EIS). Examples of connections include database connections, Java Message Service connections, and SAP R/3 connections.

As indicated in the Java EE Connector Architecture (JCA) Specification, each enterprise information system (EIS) needs a resource adapter and a connection factory. This connection factory is then accessed through the following programming model. If you use Rational Application Development tools, most of the following deployment descriptors and code are generated for you. This example shows the manual method of accessing an EIS resource.

Procedure

1. Declare a connection factory resource reference in your application component deployment descriptors, as described in this example:

```
<resource-ref>
   <description>description</description>
   <res-ref-name>eis/myConnection</res-ref-name>
   <res-type>javax.resource.cci.ConnectionFactory</res-type>
   <res-auth>Application</res-auth>
</resource-ref>
```

2. During the deployment process, configure each resource adapter and associated connection factory through the console. See the topics on installing a resource adapter and configuring a connection factory for more information.
3. Locate the corresponding connection factory for the EIS resource adapter using Java Naming and Directory Interface (JNDI) lookup in your application component, during run time.
4. Get the connection to the EIS from the connection factory.
5. Create an interaction from the connection object.
6. Create an *InteractionSpec* object. Set the function to execute in the InteractionSpec object.
7. Create a record instance for the input and output data used by function.
8. Execute the function through the Interaction object.
9. Process the record data from the function.
10. Close the connection.

Example

The following code segment shows how an application component might create an interaction and implement it on the EIS:

```
javax.resource.cci.ConnectionFactory connectionFactory = null;
javax.resource.cci.Connection connection = null;
javax.resource.cci.Interaction interaction = null;
javax.resource.cci.InteractionSpec interactionSpec = null;
javax.resource.cci.Record inRec = null;
javax.resource.cci.Record outRec = null;

try {
// Locate the application component and perform a JNDI lookup
   javax.naming.InitialContext ctx = new javax.naming.InitialContext();
   connectionFactory = (javax.resource.cci.ConnectionFactory)
ctx.lookup("java:comp/env/eis/myConnection");

// create a connection
   connection = connectionFactory.getConnection();

// Create Interaction and an InteractionSpec
   interaction = connection.createInteraction();
   interactionSpec = new InteractionSpec();
   interactionSpec.setFunctionName("GET");

// Create input record
   inRec = new javax.resource.cci.Record();

// Execute an interaction
   interaction.execute(interactionSpec, inRec, outRec);

// Process the output...

} catch (Exception e) {
   // Exception Handling
}
finally {
    if (interaction != null) {
         try {
              interaction.close();
         }
         catch (Exception e) {/* ignore the exception*/}
```

```
    }
    if (connection != null) {
         try {
              connection.close();
         }
         catch (Exception e) {/* ignore the exception */}
    }
}
```

JDBC application development tips

By using best practices to help maximize the efficiency of JDBC queries, you can potentially increase application performance.

Most of the following recommendations assume that you use DB2 on z/OS.

- Program according to the most current JDBC specifications.
- Use prepared statements to allow dynamic statement cache of DB2 on z/OS.
- Do not include literals in the prepared statements; use a parameter marker "?" to allow dynamic statement cache of DB2 on z/OS.
- Use the right getXxx method by each data type of DB2.
- Turn auto commit off when just read-only operations are performed.
- Use explicit connection context objects.
- When coding an iterator, you have a choice of named or positioned. Positioned iterators have the better performance potential.
- Close prepared statements before reusing the statement handle to prepare a different SQL statement within the same connection.
- As a bean developer, you have the choice of using JDBC or Structured Query language in Java (SQLJ) queries. JDBC makes use of dynamic SQL whereas SQLJ generally is static and uses pre-prepared plans. SQLJ requires an extra step to create and bind the plan whereas JDBC does not. SQLJ, as a general rule, is faster than JDBC.
- With JDBC and SQLJ, you are better off writing specific calls that retrieve just what you want rather than generic calls that retrieve the entire row. There is a high per-field cost.

JDBC application cursor holdability support

The cursor holdability feature can reduce the overhead of JDBC interaction with your relational database, thereby helping to increase application performance.

By activating cursor holdability, you keep a result set available across transaction boundaries for use by multiple JDBC calls. The holdability setting triggers a database cursor to keep newly updated rows active beyond the commit of the transaction that generated the new values, or result set. Hence the cursor makes the result set available for use by statements in a subsequent transaction.

Setting cursor holdability

Use one of the following techniques to set cursor holdability. For more details, see the JDBC 3.0 specification, available at the Oracle website at http://www.oracle.com/technetwork/java/index.html.

- Specify the ResultSet.HOLD_CURSORS_OVER_COMMIT parameter when creating or preparing a statement using the createStatement, prepareStatement, or prepareCall methods.
- Invoke the setHoldability method on the Connection object. The cursor holdability value that you set with this method becomes the default. If you specify cursor holdability on the Statement object, that value overrides the value that you specified on the connection.

  You cannot specify cursor holdability on a shareable connection after that connection is referenced by a second handle. Invoking the holdability method at this point generates an exception. If you want to set cursor holdability on a shareable connection, invoke the method before the connection is enlisted. Otherwise a shareable connection retains the same holdability value that applied in the previous enlistment.

- Check your database documentation to see if the product supports cursor holdability as a data source property. DB2, for example, responds to the holdability trigger if you set it as a data source custom property. See the topic, Custom property settings, for more information.

The impact of connection and transaction behaviors on cursor holdability

Setting cursor holdability in WebSphere Application Server results in the following behavior for different transaction events:

- When a connection is closed, all statements and result sets are closed even if you have set cursor holdability.
- When a transaction is rolled back, all result sets are closed even if you have set cursor holdability.
- When a local transaction is committed, both shareable and unshareable connections can have an open result set across a transaction boundary.
- When a global transaction is committed, unshareable connections can have an open result set across a transaction boundary. For shareable connections, the statements and result sets are closed even if you have set cursor holdability; the holdability value does not impact shareable connections participating in global transactions.
- When a local transaction scope ends, either at the method level or the activity session level, all statements and result sets for shareable connections are closed. Statements and result sets for unshareable connections remain open until the close method is called on the connection.

Note: For a global transaction with an unshareable connection, the backend database has responsibility for supporting cursor holdability.

Data access bean types

For easy data access programming, WebSphere Application Server provides a special class library that implements many methods of the Java Database Connectivity (JDBC) API for you. The library is essentially a set of Service Data Objects (SDO).

To make things clearer, you can refer to the classes by the name of the Java archive (JAR) file that contains them:

`databeans.jar` - This JAR file ships with WebSphere Application Server. This file contains classes that enable you to access the database using the JDBC API.

`ivjdab.jar` - This JAR file ships with Visual Age for Java. This file contains all of the classes in the `databeans.jar` file and classes that support easy use of the data access beans from the Visual Age for Java Visual Composition Editor.

`dbbeans.jar` - This JAR file ships with Rational Application Developer. This file contains a set of data access beans to more closely conform to the JDBC 2.0 RowSet standard.

The `com.ibm.db` package is provided to support existing applications that use data access beans.

IBM strongly suggests that any new applications using data access beans be developed using the `com.ibm.db.beans` package that is provided with Rational Application Developer.

Example: Using data access beans. Data access beans are essentially a class library that makes it easier to access a database. The library contains a set of beans with methods that access the database through the Java Database Connectivity (JDBC) API. This example shows using data access beans in WebSphere Application Server Version 5 and later to create new applications that use the com.ibm.db.beans package.

```
package example;
import com.ibm.db.beans.*;
import java.sql.SQLException;
```

```
public class DBSelectExample {

 public static void main(String[] args) {

  DBSelect select = null;

  select = new DBSelect();
  try {

   // Set database connection information
   select.setDriverName("COM.ibm.db2.jdbc.app.DB2Driver");
   select.setUrl("jdbc:db2:SAMPLE");
   select.setUsername("userid");
   select.setPassword("password");

   // Specify the SQL statement to be executed
   select.setCommand("SELECT * FROM DEPARTMENT");

   // Execute the statement and retrieve the result set into the cache
   select.execute();

   // If result set is not empty
   if (select.onRow()) {
    do {
     // display first column of result set
     System.out.println(select.getColumnAsString(1));
     System.out.println(select.getColumnAsString(2));
    } while (select.next());
   }

   // Release the JDBC resources and close the connection
   select.close();

  } catch (SQLException ex) {
   ex.printStackTrace();
  }
 }
}
```

Accessing data from application clients

To access a database directly from a Java Platform, Enterprise Edition (Java EE) application client, you retrieve a *javax.sql.DataSource* object from a resource reference configured in the client deployment descriptor. This resource reference is configured as part of the deployment descriptor for the client application, and provides a reference to a pre-configured data source object.

About this task

Note that data access from an application client uses the JDBC driver connection functionality directly from the client side. It does not take advantage of the additional pooling support available in the application server run time. For this reason, your client application should utilize an enterprise bean running on the server side to perform data access. This enterprise bean can then take advantage of the connection reuse and additional added functionality provided by the product run time.

Procedure

1. Import the appropriate JDBC API and naming packages:
   ```
   import java.sql.*;
   import javax.sql.*;
   import javax.naming.*;
   ```
2. Create the initial naming context:
   ```
   InitialContext ctx = new InitialContext();
   ```
3. Use the *InitialContext* object to look up a data source object from a resource reference.

```
javax.sql.DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/myDS");
//where jdbc/myDS is the name of the resource reference
```

4. Get a *java.sql.Connection* from the data source.
   - If no user ID and password are required for the connection, or if you are going to use the *defaultUser* and *defaultPassword* that are specified when the data source is created in the Application Client Resource Configuration tool (ACRCT) in a future step, use this approach:

     ```
     java.sql.Connection conn = ds.getConnection();
     ```
   - Otherwise, you should make the connection with a specific user ID and password:

     ```
     java.sql.Connection conn = ds.getConnection("user", "password");
     //where user and password are the user id and password for the connection
     ```
5. Run a database query using the *java.sql.Statement*, *java.sql.PreparedStatement*, or *java.sql.CallableStatement* interfaces as appropriate.

   ```
   Statement stmt = conn.createStatement();
   String query = "Select FirstNme from " + owner.toUpperCase() + ".Employee where LASTNAME = '" + searchName + "'";
   ResultSet rs = stmt.executeQuery(query);
   while (rs.next()) { firstNameList.addElement(rs.getString(1));
   }
   ```
6. Close the database objects used in the previous step, including any *ResultSet*, *Statement*, *PreparedStatement*, or *CallableStatement* objects.
7. Close the connection. Ideally, you should close the connection in a *finally* block of the *try...catch* statement wrapped around the database operation. This action ensures that the connection gets closed, even in the case of an exception.

   ```
   conn.close();
   ```

Service Data Objects version 2.1.1

Service Data Objects (SDO) is a framework for data application development that provides an architecture and application programming interfaces (API). The product includes an implementation of SDO 2.1.1 interfaces. After the product is installed and SDO is enabled, SDO 2.1.1 becomes the default SDO implementation when programming with SCA or SDO interfaces.

transition: For backwards compatibility, deprecated functions like JDBMediator continue to use previous SDO implementations described in “Data access with Service DataObjects, API versions 1.0 and 2.01”. Otherwise, SDO clients are bound to the SDO 2.1.1 implementation.

SDO 2.1.1 is a Java standard approved by the Java Community Process (JSR 235). For details on SDO 2.1.1, refer to the JSR 235 specification.

The product implementation complies with JSR 235, and provides some implementation-specific extensions. These extensions align with the latest direction of the SDO 3.0 specification under development at OASIS. Although there is no guarantee, these 3.0 API extensions will likely be officially standardized in future versions of SDO. Table 1 lists the 3.0 API extensions included in the product.

Table 9. SDO 3.0 API extensions included in the product. The SDO 3.0 API extensions might be standardized in future specifications.

SDO interface or class name	Method or constant	SDO 3.0 extension change
interface commonj.sdo.Type;	getHelperContext()	New SDO 3.0 method
interface commonj.sdo.helper.TypeHelper:	SDO_URI	New SDO 3.0 constant
	SDO_JAVA_URI	New SDO 3.0 constant
	SDO_XML_URI	New SDO 3.0 constant
interface commonj.sdo.Sequence:	add(String)	Removed deprecated method
	add(int, String)	Removed deprecated method
interface commonj.sdo.helper.HelperContext:	getIdentifier()	New SDO 3.0 method
class commonj.sdo.helper.HelperProvider:		Replaced SPI with SDO 3.0 version

Table 9. SDO 3.0 API extensions included in the product (continued). The SDO 3.0 API extensions might be standardized in future specifications.

SDO interface or class name	Method or constant	SDO 3.0 extension change
class commonj.sdo.helper.SDO		New API in SDO 3.0
interface commonj.sdo.helper.HelperContextFactory		New API in SDO 3.0
class commonj.sdo.impl.Environment		New SPI in SDO 3.0
class commonj.sdo.impl.Resolvable		New SPI in SDO 3.0

For information on SDO HelperContext, see “Creating and accessing SDO HelperContext”.

Creating and accessing SDO HelperContext objects:

The Service Component Architecture (SCA) implementation complies with Service Data Objects (SDO) 2.1.1 (JSR 235), and provides some implementation-specific extensions. These extensions align with the latest direction of the OASIS SDO 3.0 specification under development. One of the extensions introduces an API for creating and managing HelperContext objects, which are in the SDO class and HelperContextFactory interface.

About this task

This topic describes how to create and access SDO HelperContext in non-SCA applications.

In versions of SDO previous to 3.0, including SDO 2.1.1, there is no standard way to create HelperContext objects. SDO helper classes are accessible from the default HelperContext and are typically accessed using their corresponding INSTANCE fields, for example, TypeHelper.INSTANCE. The use of INSTANCE fields is discouraged in SDO 2.1.1, and will likely be deprecated in SDO 3.0. Instead of using INSTANCE fields, code your applications to access helpers using their corresponding accessor method on the HelperContext interface, for example, helperContext.getTypeHelper(). In SDO 2.1.1, the only HelperContext available through standard APIs is the default helper context: HelperProvider.getDefaultContext().

The proposed SDO 3.0 scoping solution, which is available in the product, is more flexible and is described in this topic.

A HelperContext represents a metadata scope in SDO. In SDO 3.0 available in the product, a HelperContext is created using a HelperContextFactory. The HelperContextFactory interface is as follows:

```
public interface HelperContextFactory {

  /**
  * Create a new HelperContext in this implementation.  Once created the HelperContext
  * can be looked up as follows (Note if the identifier is null or "" it is not registered):
  * SDO.getHelperContext(identifier);
  * @param identifier - A unique identifier that can be used to access the HelperContext.
  * @param properties - Properties required to initialize the HelperContext.
  * @return a HelperContext object
  * @throws IllegalArgumentException If a different HelperContext is already
  * registered with the specified identifier.
  */
  public HelperContext createHelperContext(String identifier, Map<String, Object> properties)
       throws IllegalArgumentException;
    /**
    * Create a new HelperContext in this implementation.  Once created the HelperContext
    * can be looked up as follows (Note if the identifier is null or "" it is not registered):
    * SDO.getHelperContext(identifier);
    * @param identifier - A unique identifier that can be used to access the HelperContext.
    * @param classLoader - The class loader for the generated static classes (if any).
    * @param properties - Properties required to initialize the HelperContext.
    * @return a HelperContext object
```

```
    * @throws IllegalArgumentException If a different HelperContext is already
    * registered with the specified identifier.
    */
  public HelperContext createHelperContext(String identifier, ClassLoader classLoader,
      Map<String, Object> properties) throws IllegalArgumentException;
}
```

There can be more than one HelperContextFactory available in an SDO environment, but one is the default. The default HelperContextFactory is accessible through the interface commonj.sdo.helper.SDO.

Procedure

1. Using the default factory, create a HelperContext object in your code.

   The following example uses the default factory to create the HelperContext `hc`:

   ```
   HelperContext hc =
   SDO.getHelperContextFactory().createHelperContext("ScopeManagerTestID", options);
   ```

   The identifier string, `"ScopeManagerTestID"`, must be unique within a Java virtual machine (JVM). If you are not concerned with the actual value, generate a guaranteed unique one using the Java UUID class, for example:

   ```
   hc = SDO.getHelperContextFactory().createHelperContext(
       UUID.randomUUID().toString(), options);
   ```

2. Access the HelperContext object in your code.

   The SDO run time manages HelperContext objects. Access an existing HelperContext object using the SDO.getHelperContext(identifier) method:

   ```
   hc = SDO.getHelperContext("ScopeManagerTestID");
   ```

   The identifier of a HelperContext can be accessed using the getIdentifier() method:

   ```
   String id = hc.getIdentifier();
   ```

Results

A HelperContext object is defined and accessible.

What to do next

Use SDO in an SCA application. When SDO is used in an SCA application, the SCA run time typically creates the HelperContext objects and identifiers. Refer to topics on using SDO 2.1.1 in SCA applications.

Because the SCA run time manages the HelperContext objects and identifiers when using SDO in SCA applications, the method used to create and access SDO HelperContext in SCA applications is different than that used in non-SCA applications. An SCA application can access SDO HelperContext using a DefaultHelperContext annotation; for example:

```
import com.ibm.websphere.soa.sca.sdo.DefaultHelperContext;

@DefaultHelperContext
public HelperContext defaultHelperContext;
```

Using SDO 2.1.1 in SCA applications:

The Service Component Architecture (SCA) implementation complies with Service Data Objects (SDO) 2.1.1 (JSR 235). You can use SDO 2.1.1 in your SCA applications.

Before you begin

Read SDO data binding for SCA applications to better understand how to work with SDO in SCA Java clients and implementations. For more information, read about using business exceptions with SCA interfaces.

Consider installing a Rational Application Developer product with SCA Development Tools that you can use to assemble service-oriented application components based on open SCA specifications. See the Rational Application Developer documentation.

About this task

SDO is supported in both the OSOA and OASIS applications. Unless otherwise specified, the information in this topic pertains to applications for both the OSOA and OASIS specifications.

To use SDO 2.1.1 in an SCA application, access the default HelperContext programmatically in a Java or Java Platform, Enterprise Edition (Java EE) component implementation type and then develop one or more SCA composites that use SDO following a bottom-up or top-down approach.

Procedure

1. Develop one or more SCA components that use SDO.
   a. Decide whether you are going to use a top-down or bottom-up approach to developing your SCA component implementations.

      The section on top-down and bottom-up development in SDO data binding for SCA applications describes the approaches. The top-down approach is typically preferred in SCA service architecture and development.
   b. Use a top-down approach (starting from WSDL and XSD files) or a bottom-up approach (starting from Java files) to develop SCA composites that use SDO.
      - Use a top-down approach to develop SCA composites.
      - Use a bottom-up approach to develop SCA composites.
   c. Access the default HelperContext programmatically.

      You can access the default HelperContext programmatically in a Java or Java EE component implementation type. You cannot access the default HelperContext programmatically in a Spring component implementation type.

   Restriction: You cannot use array types for Java parameters. You cannot map from schema elements declared with the attribute, `maxOccurs="unbounded"` to array types. Instead, map the element to a List, such as a `List<DataObject>` or `List<String>`.
2. If you are using a top-down development approach, package your WSDL and XSD files in an SCA contribution.

   Packaging WSDL and XSD files in an appropriate contribution enables your component to access the schema definitions. The section on schema registration in SDO data binding for SCA applications describes the registration.

   Consider packaging your WSDL and XSD files so that they can be accessed across various parts of your application from a shared in-memory instance. For details, see Implementing shared scopes in SCA applications that use SDO.
3. Deploy your SCA composites that use SDO in an SCA business-level application.

   See topics on SCA contributions and shared scopes for information on how to establish a shared SDO scope. Otherwise, there are no special deployment considerations for applications using the SDO data binding.

Results

You have developed and deployed an SCA composite that uses SDO in a business-level application.

What to do next

Test the deployed SCA composites.

SDO data binding for SCA applications:

Service Data Objects (SDO) is a framework for data application development that provides an architecture and application programming interfaces (API). Product support for Service Component Architecture (SCA) includes an implementation of SDO 2.1.1 interfaces. To work with SDO in SCA Java clients and implementations, it is helpful to understand SDO data binding concepts.

The product supports SDO for both OSOA and OASIS specifications, unless otherwise stated in this topic.

The SDO data binding support consists of the following concepts:
- Scope management
- Wire format serialization or deserialization
- Top-down and bottom-up development
- Schema registration
- Shared scopes
- JAX-WS based programming model

Scope management

In SDO, a scope typically corresponds to a commonj.sdo.helper.HelperContext instance. This scope sets visibility boundaries for SDO types. In the product runtime environment, the SCA layer defines HelperContext objects on SCA application boundaries, which are meaningful boundaries from an SCA perspective. Together, SCA and SDO define a default HelperContext for a given application and enable the SCA application to access the HelperContext programmatically.

In the product runtime environment, the primary SCA application scope is tied to the deployable composite. There is a 1-1 relationship between a deployable composite and an SCA default HelperContext that is managed in the runtime environment. To determine the default HelperContext of an SCA component, you must identify the deployable composite which defines this component to the SCA domain.

Wire format serialization or deserialization

An important use of SDO is to serialize or deserialize application data to and from the data format “on the wire” (the “wire format”) for the various binding configurations. The SCA runtime environment uses the default HelperContext with its registered schema definitions to perform this serialization or deserialization. This process is relevant for serializing or deserializing input argument values and return values of SCA service interfaces when the client or implementation uses the SDO data binding.

Top-down and bottom-up development

Both top-down (starting from WSDL and XSD files) and bottom-up (starting from Java files) approaches can be used with SDO in SCA applications. This section briefly explains three different usage patterns. You can combine aspects of all three patterns in a single application.

- Top-down, strongly typed

  In this usage pattern, start with a composite definition file that uses the `<interface.wsdl>` element in describing a component service interface; for example:

  ```
  <service ...>
    <interface.wsdl ...> Refers to WSDL with updateAccount
  ```

  The doc-lit-wrapped WSDL and XSD definition uses a specific type, and not `xsd:anyType`:

```
<element name="updateAccount">
  <complexType>
    <sequence>
      <element name="person" type="p:Person"/>
```

This definition maps to a Java method; for example:

```
void updateAccount(DataObject person)
```

With this usage pattern, you can write your Java code so that is reusable, even later with types other than `p:Person`. You can potentially reuse the same dynamically typed Java code with multiple strongly typed interfaces, without having to regenerate any Java code.

You can also write your Java code so that the SDO usage is tightly coupled with a specific XSD type. Further, you can mix the tightly coupled and dynamically typed programming styles in the same SCA Java application.

- Top-down, weakly typed

  In this usage pattern, start with a composite definition file that uses the `<interface.wsdl>` element; for example:

  ```
  <service ...>
    <interface.wsdl ...> Refers to WSDL with updateAccount
  ```

  The doc-lit-wrapped WSDL and XSD definition uses `xsd:anyType`:

  ```
  <element name="updateAccount">
    <complexType>
      <sequence>
        <element name="arg0" type="xsd:anyType"/>
  ```

  This definition maps to a Java method; for example:

  ```
  void updateAccount(DataObject arg0)
  ```

  This usage pattern is like top-down, strongly-typed. With the top-down, weakly typed usage pattern both WSDL and XSD files map to equivalent Java code. However, you might write your Java code to enable it to handle any type of data, as opposed to the top-down, strongly-typed usage pattern where you expect a specific type, even though you work with it through the dynamic SDO APIs.

- Bottom-up

  In this usage pattern, you start with a Java interface method such as:

  ```
  void updateAccount(DataObject arg0)
  ```

  The method maps to a WSDL or XSD definition such as:

  ```
  <element name="updateAccount">
    <complexType>
      <sequence>
        <element name="arg0" type="xsd:anyType"/>
  ```

  Depending on the bindings, you might view or work directly with the WSDL and XSD files to which the Java object maps. When mapping from Java to XSD, the SCA runtime environment maps `commonj.sdo.DataObject` to `xsd:anyType`.

  You cannot use `java.lang.Object` as a dynamic, generic interface method parameter or return type, instead of `commonj.sdo.DataObject`. The SCA runtime environment relies on introspection of the method type to signal that it uses SDO, and not JAXB, to work with this type; for example, to construct the argument values upon deserializing data on the wire.

In each of the usage patterns, you can mix styles within a single operation. For example, suppose you have the following doc-lit-wrapped WSDL definition:

```
<element name="updateAccount">
  <complexType>
    <sequence>
      <element name="arg0" type="xsd:anyType"/>
      <element name="id" type="xsd:string"/>
```

The WSDL definition maps to the following Java method:

```
void updateAccount(DataObject arg0, String id)
```

The `arg0` field is dynamically typed by DataObject, while the `id` field is statically typed by String.

Schema registration

The SCA runtime environment provides mechanisms to register schema definitions from your SCA application (for example, in WSDL and XSD files packaged with your application) to the default HelperContext of your SCA application. All schema definitions packaged within the same contribution Java archive (JAR) file as the one your deployable composite is contributed within are registered with the default HelperContext of that deployable composite.

Schema registration is important because it affects the exact details of your application SDO programming model. For example, the `DataFactory.create()` methods can be used to create DataObject instances of the corresponding schema definitions that are registered in the runtime environment, without having to reference or load the XSD files containing these schema definitions in application code.

Some styles of interface definition and programming result in DataObject instances of unknown type. Within the SDO type system, the XSD type of such an object is the SDO-equivalent of `xsd:anyType`. Methods might return different values depending on whether the instance is of a known, specific type or the `anyType`-equivalent.

For example, SDO API calls such as the following might return different results depending on whether the object instances are recognized in the SDO type system as instances of a registered XSD type. For the following call, `get(int propertyIndex)` returns a List for `xsd:anyType`:

```
dataObjectInstance.get(1);
```

The following call returns an empty List for `xsd:anyType`:

```
dataObjectInstance.getType().getDeclaredProperties();
```

Shared scopes

Shared scopes are only supported for applications based on OSOA specifications.

SCA contribution support enables convenient packaging of WSDL and XSD definition files used by multiple SCA application composites using the contribution import and export mechanism. The common WSDL and XSD files can be packaged into a single contribution JAR file, which is like a shared library.

The SDO data binding function further uses the SCA contribution support by enabling you to establish a shared scope (HelperContext) for your SDO types. This reduces the memory footprint involved in loading schema definitions from large WSDL and XSD files used across your application and in constructing the corresponding SDO Type definitions from them.

The shared scope is maintained at the level of a business-level application. For any business-level application, all references to an SDO Type that correspond to a schema definition from an XSD or WSDL file imported from a shared contribution resolve to a single SDO Type instance in a single scope (HelperContext).

For shared scopes, there are two restrictions:

- A contribution that exports schema definitions cannot, in turn, import other schema definitions from another contribution.
- You cannot divide namespaces across more than one contribution. That is, you cannot package schema definitions in a given target namespace in one contribution and then import others in that same target namespace from a second contribution.

JAX-WS based programming model

In the product, the mapping between a WSDL operation and Java method is defined by JAX-WS. Within that operation-level mapping, it is the mapping between specific XSD types and the corresponding Java parameter types that is defined by the particular data binding (SDO in this case, rather than JAXB). Other than the type mapping, the programming model is independent of the choice of data binding.

One consequence of this capability is that the JAX-WS annotations such as @RequestWrapper, @ResponseWrapper, @WebParam, @WebResult are significant in SCA applications using SDO.

Another important consequence is that the product uses JAX-WS to define the mapping between a Java exception that occurs in Java clients and implementations, and the fault bean that is serialized on the wire. Thus, the fault bean can be an SDO (of type commonj.sdo.DataObject). In which case, SDO is used to serialize or deserialize the fault bean to or from the wire format.

Using a top-down approach to develop SCA components that use SDO:

You can use a top-down approach that starts from Web Services Description Language (WSDL) or XML definition files to develop Service Component Architecture (SCA) component implementations that use Service Data Objects (SDO) 2.1.1 (JSR 235).

Before you begin

Consider installing a Rational Application Developer product with SCA Development Tools that you can use to assemble service-oriented application components based on open SCA specifications. See the Rational Application Developer documentation.

Access the default HelperContext programmatically in a Java or Java Platform, Enterprise Edition (Java EE) component implementation type. Complete step 1 of Using SDO 2.1.1 in SCA applications.

About this task

This topic describes how to develop SCA composites that use SDO following a top-down approach.

Unless otherwise specified, this topic describes how to develop both OSOA and OASIS SCA composites that use SDO.

Procedure

1. Describe the service interface in your WSDL and XSD definition files.

   The following example WSDL and XSD files describe a service interface.

   Example WSDL file, test.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://test" xmlns:tns="http://test"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="Test">

  <wsdl:types>
    <schema targetNamespace="http://test" xmlns="http://www.w3.org/2001/XMLSchema" xmlns:p="http://person">
      <import namespace="http://person" schemaLocation="person.xsd"/>

      <element name="updateAccount">
        <complexType>
          <sequence>
            <element name="person" type="p:Person"/>
            <element name="code" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
```

```
      <element name="updateAccountResponse">
        <complexType>
          <sequence>
            <element name="response" type="p:Status"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>

  <wsdl:message name="PersonRequestMessage">
    <wsdl:part element="tns:updateAccount" name="parameters"/>
  </wsdl:message>

  <wsdl:message name="PersonResponseMessage">
    <wsdl:part element="tns:updateAccountResponse" name="parameters"/>
  </wsdl:message>

  <wsdl:portType name="Test">
    <wsdl:operation name="updateAccount">
      <wsdl:input message="tns:PersonRequestMessage" name="ReqMsgName"/>
      <wsdl:output message="tns:PersonResponseMessage" name="RespMsgName"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="TestSoapBinding" type="tns:Test">
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="updateAccount">
      <wsdlsoap:operation soapAction="urn:updateAccount"/>
      <wsdl:input name="ReqMsgName">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="RespMsgName">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="TestService">
    <wsdl:port binding="tns:TestSoapBinding" name="TestSoapPort">
      <wsdlsoap:address location=""/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Example XSD file, person.xsd, that test.wsdl imports

```
<?xml version="1.0" encoding="UTF-8"?>

  <schema targetNamespace="http://person"  xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="Person">
      <sequence>
        <element name="firstName" type="string"/>
        <element name="lastName" type="string"/>
      </sequence>
    </complexType>

    <complexType name="Status">
      <sequence>
        <element name="statusCode" type="int"/>
        <element name="message" type="string"/>
      </sequence>
    </complexType>
  </schema>
```

2. Produce a corresponding Java interface.

   If you are using an IBM Rational tool that supports SCA, you do not need to complete this step. The tool generates the code for you. Proceed to step 3.

   If you are not using a Rational tool that supports SCA, you can manually produce a Java interface from a WSDL file by completing this step 2, which provides an example that modifies generated code. There is not an automated tool to produce a Java interface from a command-line interface.

   a. Identify the WSDL and XSD files from which you want to produce Java interfaces.

      This example uses the `test.wsdl` and `person.xsd` files in step 1.

   b. Run the **wsimport** command with the `-s` option to save source files.

      ```
      wsimport.bat -s . test.wsdl
      ```

c. Among the generated Java files, identify the SEI.

   The SEI is a Java interface, and not a class file, with a class-level @WebService annotation, javax.jws.WebService.

   In this example, the Java interface is the `test/Test.java` file. Running the **wsimport** command produces the following output:

```
//
// Generated By:JAX-WS RI, IBM 2.1.1 in JDK 6 (JAXB RI, IBM JAXB 2.1.3 in JDK 1.6)
//

package test;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;
import person.Person;
import person.Status;

@WebService(name = "Test", targetNamespace = "http://test")
@XmlSeeAlso({
    person.ObjectFactory.class,
    test.ObjectFactory.class
})
public interface Test {

    /**
     *
     * @param person
     * @param code
     * @return
     *     returns person.Status
     */
    @WebMethod(action = "urn:updateAccount")
    @WebResult(name = "response", targetNamespace = "")
    @RequestWrapper(localName = "updateAccount", targetNamespace =
            "http://test", className = "test.UpdateAccount")
    @ResponseWrapper(localName = "updateAccountResponse", targetNamespace =
            "http://test", className = "test.UpdateAccountResponse")
    public Status updateAccount(
        @WebParam(name = "person", targetNamespace = "")
        Person person,
        @WebParam(name = "code", targetNamespace = "")
        String code);
}
```

d. Modify the SEI.

   1) For all parameter and return values of generated (JAXB) types, change the Java type to `commonj.sdo.DataObject`. All complex schema types must map to `commonj.sdo.DataObject` Java types. Also, remove all imports of these generated JAXB types.

   2) For each @RequestWrapper and @ResponseWrapper annotation, change the value of the `className` element to `commonj.sdo.DataObject`. Otherwise, leave the JAX-WS annotations because they are significant.

   3) Remove or comment out the @XmlSeeAlso block.

Completing these steps results in the following Java interface:

```
//
// Generated By:JAX-WS RI, IBM 2.1.1 in JDK 6 (JAXB RI, IBM JAXB 2.1.3 in JDK 1.6)
//

package test;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
```

```
import javax.xml.ws.ResponseWrapper;
import commonj.sdo.DataObject;

@WebService(name = "Test", targetNamespace = "http://test")

public interface Test {

    /**
     *
     * @param person
     * @param code
     * @return
     *     returns person.Status
     */
    @WebMethod(action = "urn:updateAccount")
    @WebResult(name = "response", targetNamespace = "")
    @RequestWrapper(localName = "updateAccount", targetNamespace =
            "http://test", className = "commonj.sdo.DataObject")
    @ResponseWrapper(localName = "updateAccountResponse", targetNamespace =
            "http://test", className = "commonj.sdo.DataObject")
    public DataObject updateAccount(
        @WebParam(name = "person", targetNamespace = "")
        DataObject person,
        @WebParam(name = "code", targetNamespace = "")
        String code);
}
```

The following example shows the resulting code with JAX-WS annotations removed for readability only. Do not remove the annotations before compiling to use the example code.

```
package test;
import commonj.sdo.DataObject;
public interface Test {
    public DataObject updateAccount(DataObject person, String code);
}
```

3. Write your SCA Java client or component implementation using the dynamic SDO programming model.

   The following example code shows a service implementation in Java. Read the source comments carefully to see the differences between OSOA and OASIS.

```
package test.impl;

import test.Test;

// FOR OSOA
import org.osoa.sca.annotations.Service;
// FOR OASIS, commented out
// import org.oasisopen.sca.annotation.Service;

import commonj.sdo.DataObject;
import commonj.sdo.helper.DataFactory;
import commonj.sdo.helper.HelperContext;

import com.ibm.websphere.soa.sca.sdo.DefaultHelperContext;

@Service(Test.class)
public class TestImpl implements Test {

    @DefaultHelperContext
    protected HelperContext myDefaultHC;

    public DataObject updateAccount(DataObject person, String code) {
        String error_msg = null;
        if (code.equals("ERROR_STRING_IN_FIELD1")) {
            error_msg = "ERROR firstName: " +  person.getString("firstName"0);
            HelperContext defaultHC = ContextHelper.getCurrentHelperContext();
            DataFactory dataFactory = myDefaultHCdefaultHC.getDataFactory();
            DataObject returnDO = dataFactory.create("http://person","Status");
            returnDO.setInt("statusCode", -1);
            returnDO.setString("message", error_msg);
            return returnDO;
        } else {
            // process(person);
```

```
            ...
        }
    }
}
```

The input `person` DataObject is of type `{http://person}Person`. The returned output DataObject is of type `{http://person}Status`. Both the input and output are registered in the SCA application default HelperContext that is accessed using the `myDefaultHC` object.

4. In a composite definition, declare your component reference or component service interface with an `<interface.wsdl>` element that refers to the original `portType` value.

   Example accountTest.composite file for OSOA

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://www.ibm.com/test/soa/sca/sdo/"
    name="AccountTestComposite">
  <component name="AccountTestComponent">
    <implementation.java class="test.impl.TestImpl"/>
    <service name="Test">
      <interface.wsdl interface="http://test#wsdl.interface(Test)"/>
    </service>
  ...
  </component>
</composite>
```

   Example accountTest.composite file for OASIS

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
    targetNamespace="http://www.ibm.com/test/soa/sca/sdo/"
    name="AccountTestComposite">
  <component name="AccountTestComponent">
    <implementation.java class="test.impl.TestImpl"/>
    <service name="Test">
      <interface.wsdl interface="http://test#wsdl.interface(Test)"/>
    </service>
  ...
  </component>
</composite>
```

5. Package the composite and deploy the component with the authored implementation along with the WSDL or XSD files into a single contribution JAR file.

   Suppose the example files are packaged into a single file, `MyAccountTestContribution.jar`. A listing of the JAR file contents from the command line is as follows:

```
$ jar tf MyAccountTestContribution.jar

wsdl/test.wsdl
wsdl/person.xsd
test/Test.class
test/impl/TestImpl.class
META-INF/accountTest.composite
META-INF/sca-contribution.xml
```

Results

You have developed an SCA composite that uses SDO following a top-down approach.

What to do next

Optionally, implement shared scopes. See the topic on using SDO 2.1.1 in SCA applications.

Deploy your files that use SDO in an SCA business-level application.

Using a bottom-up approach to develop SCA components that use SDO:

You can use a bottom-up approach that starts from Java files to develop Service Component Architecture (SCA) component implementations that use Service Data Objects (SDO) 2.1.1 (JSR 235).

Before you begin

Consider installing a Rational Application Developer product with SCA Development Tools that you can use to assemble service-oriented application components based on open SCA specifications. See the Rational Application Developer documentation.

Access the default HelperContext programmatically in a Java or Java Platform, Enterprise Edition (Java EE) component implementation type. Complete step 1 of Using SDO 2.1.1 in SCA applications.

About this task

This topic describes how to develop SCA composites that use SDO following a bottom-up approach.

Procedure

1. Start from a Java interface or implementation using type commonj.sdo.DataObject for one or more parameters or return types.

   The following example Java file provides an interface.

   Logger.java (interface):

   ```
   package logger;
   import commonj.sdo.DataObject;
   public interface Logger {
       public String logDataObjectProperties (DataObject input);
   }
   ```

2. Write your SCA Java client or component implementation using the dynamic SDO programming model.

   The following example Java file provides a component implementation.

   LoggerImpl.java (component implementation):

   ```
   package logger.impl;

   import logger.Logger;
   import org.osoa.sca.annotations.Service;

   import commonj.sdo.DataObject;
   import commonj.sdo.helper.DataFactory;
   import commonj.sdo.helper.HelperContext;

   @Service(Logger.class)
   public class LoggerImpl implements Logger {

       public String logDataObjectProperties (DataObject input) {
           String logMsg =  "==========\n";
           List props = input.getInstanceProperties();
           for (int i=0; i < props.size(); i++){
               Property prop = (Property)props.get(i);
               logMsg += "  prop[" + i + "], name = " + prop.getName() + ", val = " +
                   input.get(prop).toString() + "\n";
           }
           logMsg += "==========\n";
           return logMsg;
       }
   }
   ```

   The SDO application programming interfaces used are generic in that they do not depend on any particular schema definitions. This behavior fits the bottom-up approach because, without a WSDL interface describing this service, the runtime environment cannot associate the input object that is built during deserialization with a specific XSD or SDO type.

Results

You have developed an SCA composite that uses SDO following a bottom-up approach.

What to do next

Optionally, implement shared scopes. See the topic on using SDO 2.1.1 in SCA applications.

Deploy your files that use SDO in an SCA business-level application.

Accessing default HelperContext objects in SCA applications:

A Service Component Architecture (SCA) application can access a Service Data Objects (SDO) 2.1.1 HelperContext object. This object either uses DefaultHelperContext annotation or implements a application programming interface (API) that uses the commonj.sdo.helper.SDO class.

Before you begin

Read SDO data binding for SCA applications to better understand how to work with SDO in SCA Java clients and implementations.

Develop one or more SCA composites that use SDO following a top-down or bottom-up approach.

About this task

This topic describes how to create and access SDO HelperContext in SCA applications. For information about accessing SDO HelperContext in non-SCA applications, see Creating and accessing SDO HelperContext objects.

You can create and access SDO HelperContext in both OSOA and OASIS SCA applications.

Because the SCA run time manages the HelperContext objects and identifiers when using SDO in SCA applications, the method used to create and access SDO HelperContext in SCA applications is different from that used in non-SCA applications. An SCA application can access SDO HelperContext using a DefaultHelperContext annotation, `@DefaultHelperContext`.

Alternatively, an SCA application can implement an API that uses the commonj.sdo.helper.SDO class, to obtain the same SCA-managed HelperContext instance. This approach is an alternative to the annotation. To use this approach, pass the String ID of the SCA-managed HelperContext, which is `sca-default`, into the SDO.getHelperContext method.

You can access the default HelperContext programmatically in a Java or Java Platform, Enterprise Edition (Java EE) component implementation type using either annotation injection or an API. You cannot access the default HelperContext programmatically in a Spring component implementation type. When you use an OSGI application as an implementation of an SCA component, you can access the default HelpContext instance using the API mechanism. However, you cannot access the instance using the annotation injection mechanism.

Procedure

1. Add a public or protected field or setter method of the `commonj.sdo.helper.HelperContext` type to your Java implementation class.

   The method can either be a Java component implementation, such as `<implementation.java>`, or a Java EE implementation class, such as an EJB implementation class.
2. Annotate the field or setter method with `@com.ibm.websphere.soa.sca.sdo.DefaultHelperContext`.

   The following example shows a field annotation:

   ```
   @DefaultHelperContext
   protected HelperContext myDefaultHC;
   ```

   The following example shows an annotation of a setter method:

```
private HelperContext helperContext;

@DefaultHelperContext
public void setHelperContext(HelperContext hc) {
   this.helperContext = hc;
}
```

Alternative step: The following example uses the API rather than annotation:

```
import commonj.sdo.helper.SDO;
import com.ibm.websphere.sdox.SDOUtil;

HelperContext helperContext = SDO.getHelperContext("sca-default")
// Or the following line is equivalent to using the string value directly
HelperContext helperContext = SDO.getHelperContext(SDOUtil.SCA_DEFAULT_SCOPE);
```

3. Use the injected HelperContext in your implementation logic.

   When your component starts, the container will inject this field, or call this setter, with the default HelperContext instance for this component so you can use it in your implementation.

```
import com.ibm.websphere.soa.sca.sdo.DefaultHelperContext;
import commonj.sdo.helper.HelperContext;

// FOR OSOA
import org.osoa.sca.annotations.Service;
// FOR OASIS, commented out
// import org.oasisopen.sca.annotation.Service;
...

// This is a Java implementation of an SCA component
@Service(AccountService.class)
public class AccountServiceImpl implements AccountService {

    private HelperContext helperContext;

    @DefaultHelperContext
    public void setHelperContext(HelperContext hc) {
        this.helperContext = hc;
    }

    @Override
    public DataObject accountMethod(DataObject account, String name) {
        // ....

        // Get dataFactory to create return object
        DataFactory dataFactory = this.helperContext.getDataFactory();
        DataObject retVal = dataFactory.create("http://myns", "Response");

        retVal.set(..)    // ... Set properties on return object

        return retVal;
    }
```

Results

You have written code that accesses the default HelperContext.

What to do next

Develop one or more SCA composites that use SDO following a bottom-up or top-down approach.

Implementing shared scopes in SCA applications that use SDO:

You can package your WSDL and XSD files so they can be accessed across various parts of a Service Component Architecture (SCA) application that uses Service Data Objects (SDO) 2.1.1.

Before you begin

Develop one or more SCA composites that use SDO following a top-down approach.

If you are using a top-down development approach, you must package WSDL and XSD files in an appropriate SCA contribution so the component can access the schema definitions. Read the section on schema registration in SDO data binding for SCA applications for details.

About this task

This topic discusses how to package WSDL and XSD files so that they can be accessed across various parts of your application from a shared in-memory instance. The packaging establishes a shared SDO scope. Shared SDO scope is only supported for applications developed on OSOA specifications.

Procedure

1. Package the WSDL and XSD files that have schema definitions which you want to share into a shared contribution JAR file.
2. In the `META-INF/sca-contribution.xml` file within the shared contribution JAR file, export the namespaces for which you want to share schema definitions within SDO scopes.

   To export the namespaces, use the `<export namespace="...">` statement.

   The examples in this topic continue the example scenario described in the topic on using SDO 2.1.1 in SCA applications.

   Example META-INF/sca-contribution.xml for mySharedDefs.jar:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
       targetNamespace="http://www.ibm.com/test/soa/sca/sdo/scope/">
     <export namespace="http://www.ibm.com/test/soa/sca"/>
     <export namespace="http://www.ibm.com/test/soa/sca/person"/>
     <export namespace="http://www.ibm.com/test/soa/sca/address"/>
     ...
   </contribution>
   ```

3. For composite applications that reference the shared WSDL and XSD schema definitions, import the desired namespaces from the appropriate `META-INF/sca-contribution.xml` file.

   Import namespaces from the `sca-contribution.xml` file of the contribution JAR that contributes the deployable composite which deploys the application components that reference the shared schema definitions as SDO definitions. To import the namespaces, use the `<import namespace="...">` statement.

   Example META-INF/sca-contribution.xml for myImportingComposite1.jar:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
       targetNamespace="http://www.ibm.com/test/soa/sca/sdo/scope/"
       xmlns:sdoscope="http://www.ibm.com/test/soa/sca/sdo/scope/">

     <deployable composite="sdoscope:MyShippingComposite"/>

     <import namespace="http://www.ibm.com/test/soa/sca"/>
     <import namespace="http://www.ibm.com/test/soa/sca/person"/>
     <import namespace="http://www.ibm.com/test/soa/sca/address"/>
     ...
   </contribution>
   ```

   Example META-INF/sca-contribution.xml for myImportingComposite2.jar:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
       targetNamespace="http://www.ibm.com/test/soa/sca/sdo/scope/"
       xmlns:sdoscope="http://www.ibm.com/test/soa/sca/sdo/scope/">

     <deployable composite="sdoscope:MyLoggingComposite"/>

     <import namespace="http://www.ibm.com/test/soa/sca"/>
     <import namespace="http://www.ibm.com/test/soa/sca/person"/>
     <import namespace="http://www.ibm.com/test/soa/sca/address"/>
     ...
   </contribution>
   ```

Results

The packaging establishes a shared SDO scope.

Assuming the composites `MyShippingComposite` and `MyLoggingComposite` are deployed using the same business-level application, then all SDO references to schema definitions in the namespaces exported from the `mySharedDefs.jar` file (and imported from the respective JAR files contributing these two composites) are scoped to a common, shared scope associated with the `mySharedDefs.jar` file. If sharing has not been established through this mechanism, the schema definitions are created separately in each of the two separate scopes associated with the each of the deployable composites.

What to do next

Deploy your files that use SDO in an SCA business-level application.

Data access with Service DataObjects, API versions 1.0 and 2.01

The Service Data Objects (SDO) framework is a data-centric, disconnected, XML-integrated, data access mechanism that provides a source-independent result set.

- SDO is data-centric because it eliminates the need for client applications to work with special formats of data, such as the object representations of the Enterprise JavaBeans (EJB) API. Instead, clients work with easily traversable graphs of DataObjects.
- SDO is disconnected because the retrieved result is independent of any back-end data store connections or transactions.
- SDO is XML-integrated in that it provides services to easily convert retrieved data to and from XML format.

Put simply, SDO is a framework for data application development, which includes an architecture and API. SDO does the following:

- Simplifies the Java Platform, Enterprise Edition (Java EE) data programming model.
- Abstracts data in a service-oriented architecture (SOA).
- Unifies data application development.
- Supports and integrates XML.
- Incorporates Java EE patterns and best practices.

The Service Data Objects framework provides a unified framework for data application development. With SDO, you do not need to be familiar with a technology-specific API in order to access and use data. You need to know only one API, the SDO API, which lets you work with data from multiple data sources, including relational databases, entity EJB components, XML pages, web services, the Java Connector Architecture, JavaServer Pages, and more.

Unlike some of the other data integration models, SDO does not stop at data abstraction. The SDO framework also incorporates a good number of Java EE patterns and best practices, making it easy to incorporate proven architecture and designs into your applications. For example, most web applications today are not (and cannot) be connected to backend systems 100 percent of the time; so SDO supports a disconnected programming model. Likewise, many applications tend to be remarkably complex, comprising many layers of concern. How will data be stored? Sent? Presented to users in a GUI framework? The SDO programming model prescribes patterns of usage that allow clean separation of each of these concerns.

SDO components

An architectural overview of SDO describes each of the components that make up the framework and explains how they work together. The first three components listed are “conceptual” features of SDO: They do not have a corresponding interface in the API.

SDO clients

SDO clients use the SDO framework to work with data. Instead of using technology-specific APIs and frameworks, they use the SDO programming model and API. SDO clients work on SDO DataObjects and do not need to know how the data they are working with is persisted or serialized.

Data mediator services

Data mediators services (DMS) are responsible for creating a DataGraph from data sources, and updating data sources based on changes made to a DataGraph. (A DataGraph is an envelope object that contains service data objects.)

The DMS provides the mechanism to move data between a client and a data source. It is created with back end specific metadata. The metadata defines the structure of the DataGraph that is produced by the DMS as well as the query to be used against the back end. When the DMS is requested to produce a DataGraph, it queries its targeted back end and transforms the native result set into the DataGraph format. Once the DataGraph is returned, the DMS no longer has any reference to it, making it stateless with respect to the DataGraph. When the DMS is requested to flush modifications of an existing DataGraph to the back end, it extracts the changes made from the original state of the DataGraph and flushes those changes to the back end. A DMS typically employs some form of optimistic concurrency control strategy to detect update collisions.

WebSphere Application Server provides functionality for two separate Data Mediator Services. If you simply need to retrieve data from a relational data source and return a DataGraph, using the Java Database Data Mediator Service is a good choice. However, if you have business logic, then you probably want an object-oriented (OO) rendering of the data into entity beans. One could consider SDO as an object rendering of data like entity beans. But entity beans have better Object-Relational (OR) mapping tools, and the EJB container and persistence manager for entity beans offer more sophisticated caching policies. Your best choice then is the EJB Data Mediator Service. The EJB mediator can work with these caches. Also, the entity bean programming model is a single level store model. You can navigate from entity to entity and the container and persistence manager either prefetches or lazily fetches in data as needed. On update, the programmer commits the transaction and the container and persistence manager do the work of tracking updated beans and writing them back to the data store and in memory cache.

Data sources

Data sources are not restricted to backend data sources (for example, persistence databases). A data source contains data in its own format. As to the SDO 1.0 API, only the DMS accesses data sources; SDO applications do not. The applications only work with SDO 1.0 DataGraphs.

Each of the following components corresponds to a Java interface in the SDO programming model.

DataObjects

As the fundamental components of SDO, DataObjects provide a common view of structured data for SDO clients. DataObjects can hold multiple different attributes of any serializable type (such as string or integer); more complex DataObjects can also contain simpler DataObjects. DataObjects hold all of their data in properties.

SDO version 1.0 DataObjects are always linked together and contained in DataGraphs. The version 1.0 DataObject interface provides simple creation and deletion methods (createDataObject() with various signatures and delete()), and reflective methods to get their types (instance class, name, properties, and namespaces). The interface also supports static object types that you create from external code generators. See the article “Dynamic and static object types for the JDBC DMS” for more information.

DataGraphs

A DataGraph is a structured result returned in response to a service request. The DMS transforms the native backend query results into the DataGraph, which is independent of the originating backend data store. This makes the DataGraph easily transferable between different data sources. The DataGraph is composed of interconnected nodes, each of which is an SDO DataObject. It is

independent of connections and transactions of the originating data source. The DataGraph tracks changes made to it from its original source. This change history can be used by the DMS to reflect changes back to the original data source. DataGraphs can easily be converted to and from XML documents enabling them to be transferred between layers within a multi-tiered system architecture. A DataGraph can be accessed in either breadth-first or depth-first manner, and it provides a disconnected data cache that can be serialized for web services

The DataGraph returned by the mediator can contain either dynamic or generated static DataObjects. Use of generated classes gives type safe interfaces for easier programming and better runtime performance. The EMF generated classes must be consistent in name and type with the schema that would be created for dynamic DataObjects except that additional attributes and references can be defined. Only those attributes and references specified in the query are filled in with data. Remaining attributes and references are not set.

Change summary

SDO 1.0 change summaries are contained by DataGraphs and are used to represent the changes that have been made to a DataGraph returned by the DMS. They are initially empty (when the DataGraph is returned to a client) and populated as the DataGraph is modified. Change summaries are used by the DMS at backend update time to apply the changes back to the data source. They enable the DMS to efficiently and incrementally update data sources by providing lists of the changed properties (along with their old values) and the created and deleted DataObjects in the DataGraph. Information is added to the change summary of a DataGraph only when the change summary's logging is activated. Change summaries provide methods for DMS to turn logging on and off.

Note: The SDO 1.0 change summary is not a client API; it is used only by the DMS.

Properties, types, and sequences

DataObjects hold their contents in a series of properties. Each property has a type, which is either an attribute type such as a primitive (for example, int) or a commonly used data type (for example, Date) or, if a reference, the type of another DataObject. Each DataObject provides read and write access methods (getters and setters) for its properties. Several overloaded versions of these accessors are provided, allowing the properties to be accessed by passing the property name (String), number (int), or property metaobject itself. The String accessor also supports an XPath-like syntax for accessing properties. For example, you can call get(“department[number=123]”) on a company DataObject to get its first department whose number is 123. Sequences are more advanced. They allow order to be preserved across heterogeneous lists of property-value pairs.

For more introductory information

For a good introduction to SDO 1.0 that also includes a small sample application, refer to the IBM developerWorks® paper “Introduction to Service DataObjects.”

Note: To fully understand the EJB data mediator service you need a good understanding of the EJB programming model. For more information refer to the articles “Task overview: Using enterprise beans in applications” and “Service Data Objects: Resources for learning.”

Java DataBase Connectivity Mediator Service:

The Java Database Connectivity (JDBC) Data Mediator Service (DMS) is the Service Data Objects (SDO) component that connects to any database that supports JDBC connectivity. It provides the mechanism to move data between a DataGraph and a database.

A regular JDBC call returns a result set in a tabular format. This format does not directly correspond to the object-oriented data model of Java, and can complicate navigation and update operations. When a client sends a query for data through the JDBC DMS, the JDBC result set of tabular data is transformed into a

DataGraph composed of related DataObjects. This enables clients to navigate through a graph to locate relevant data rather than iterating through rows of a JDBC result set. Once you have altered the DataGraph, all of the changes can be committed together and propagated back to the database by the JDBC DMS. Between the processes of being populated and committed, the DataGraph is disconnected from the database, and there are no locks held on the data accessed. Once disconnected allows multiple changes to be made to the graph without making additional round trips to the database, improving performance.

The JDBC DMS is created with server-specific metadata. The metadata defines the structure of the DataGraph that is produced by the DMS, as well as the query to be used against the server.

Metadata for Data Mediator Service:

A Data Mediator Service (DMS) is the Service Data Object (SDO) component that connects to the back end database. It is created with back end specific metadata. The metadata defines the structure of the DataGraph that is produced by the DMS as well as the query to be used against the back end.

Metadata is composed of the following components:

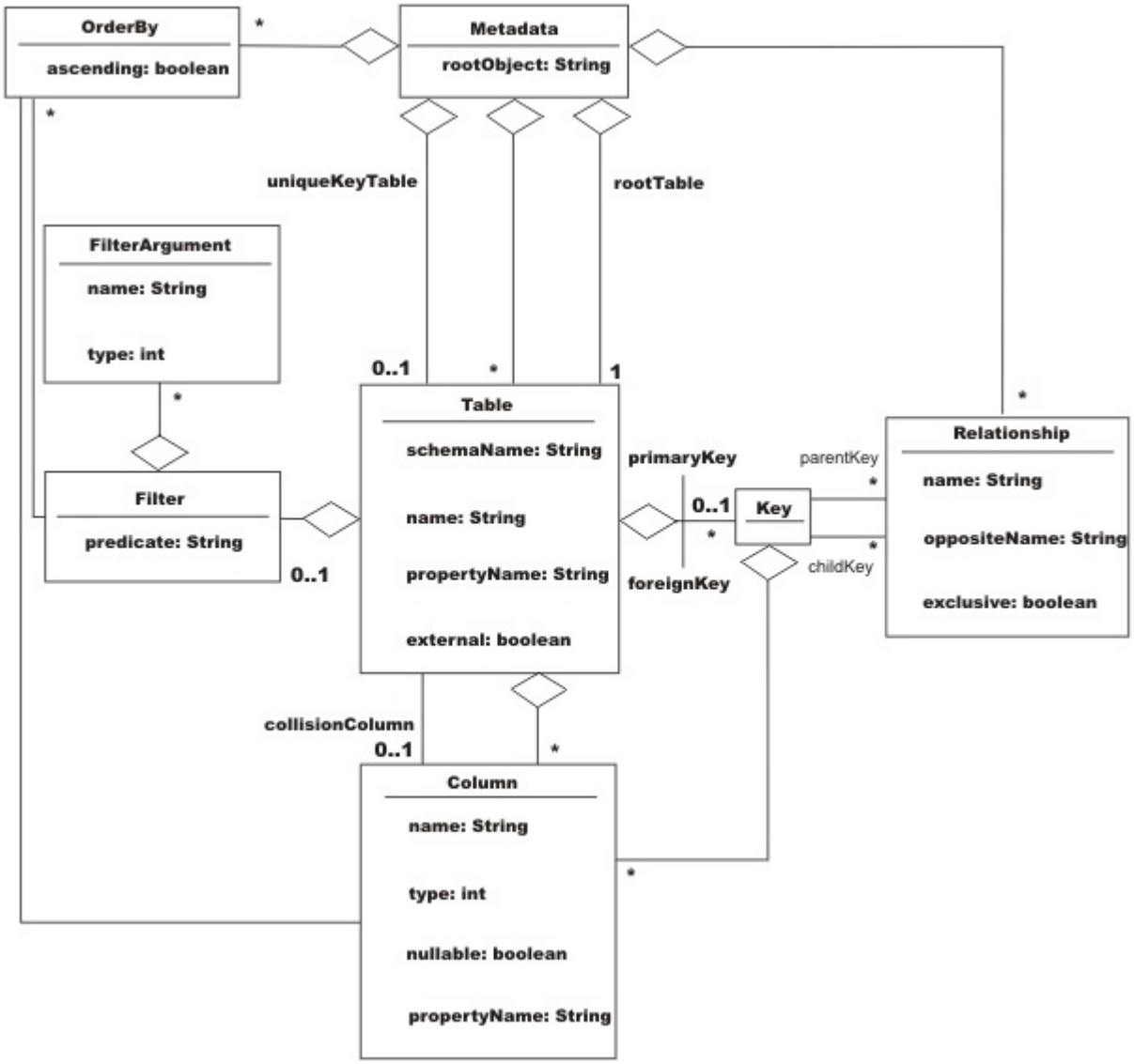


Table This represents a table within the target database and is composed of the following items:

Name This is the database table name. A table might also have a property name that can be

used to specify the name of the DataObject that corresponds to this table. By default, the property name is the same as the table name.

Columns
The subset of database table columns to return from the database. A column has a type that corresponds to a JDBC type and it can prohibit null entries. A column has a name that corresponds to the name in the database and an optional property name that identifies the column name in the DataObject. By default, the property name is the same as the column name in the database.

Primary Key
The column (or columns) used to uniquely identify a row within the table.

Note: Keys may be composed of multiple columns. The following example illustrates creation of a compound primary key :

```
Key pk = MetadataFactory.eINSTANCE.createKey();
pk.getColumns().add(xColumn);
pk.getColumns().add(yColumn);
coordinateTable.setPrimaryKey(pk);
```

If a table is related to this one and is the child table, it uses the same method to create the foreign key to point to this coordinate table.

Foreign Key
The column (or columns) used to relate the table to another table in the metadata. There is an assumed positional mapping between compound primary keys and foreign keys. For example, if a parent table has a primary key such as (x,y) with respective types (integer, string), then it is expected that any pointing foreign key is also (x', y') with respective types (integer, string).

Note: Keys may be composed of multiple columns. The following example illustrates the creation of a compound foreign key :

```
Key fk = MetadataFactory.eINSTANCE.createKey();
fk.getColumns().add(xColumn);
fk.getColumns().add(yColumn);
coordinateTable.getForeignKeys().add(fk);
```

If a table is related to this one and is the child table, it uses the same method to create the foreign key to point to this coordinate table.

Filter A structured query language (SQL) WHERE clause predicate that can be given with or without parameters to fill in later. This is added to the DataGraph SELECT statement WHERE clause. It is not parsed or interpreted in any way; it is used as is. If given with parameters to fill in later, these parameters become arguments passed into the JDBC DMS when getting the DataGraph. Filters are used with generated queries only. If a supplied query is given, the metadata filters are ignored.

Relationship
Relates two tables through the primary key of the parent table and the foreign key of the child table. Relationships are composed of the following items:

Name This is the name given to the relationship, usually associated with how the two tables are related. If *Customers* is the parent table and *Orders* is the child table, then the default name of the relationship is Customers_Orders.

Opposite Name
This is the name used to navigate from the child DataObject to the parent DataObject.

Parent Key
The primary key of the parent table.

Child Key
The foreign key of the child table that points to the parent key.

Exclusive
By default, a Relationship causes the generated query to use an inner join operation on the two tables involved in the relationship. This means that it only returns the parent entries that have children, that is, child entries pointing to them. If the value of the Exclusive attribute is set to false, the query uses a left outer join operation instead and returns all parent entries, even those without children.

Ordering
Columns used for ordering the tables. Can be either ascending or descending. When specified, this causes generated queries to contain an ORDER BY clause.

Dynamic and static object types for the JDBC DMS:

DataObjects of the Service Data Object (SDO) 1.0 Specification can use static types as well as dynamic types. If you know that a particular dataGraph schema meets all of your application query requirements, you can generate static SDO code for potential runtime benefits.

With dynamic types, the information that defines the shape of a DataGraph is constructed at runtime. The DataGraph schema is created by the JDBC data mediator service (DMS) from the metadata provided upon creation. The JDBC DMS only requires the metadata and a connection to a data source to produce the DataGraph with dynamic typing. This is the default method for creating the JDBC DMS.

If you know the shape of the DataGraph at development time, you can use a code generator to create strongly typed interfaces (static data API code) that simplify DataGraph navigation, provide better compile-time checking for errors, and improve performance. For more information about the metamodels from which you can generate static SDO code, consult the introduction of the SDO 1.0 Specification. The introduction contains a list of the specification scope requirements, where you can find a brief discussion on support for static data API. Note that the dynamic API is still available when you use strongly typed DataObjects.

With the code generator you create classes for each DataObject type in the DataGraph. Each class contains `getter()` and `setter()` methods for each property in the DataObject. This enables a client to call type-safe methods rather than passing in the name of a property. For example, instead of calling the property `DataObject.get("CUSTFIRSTNAME")`, the generated types can contain a DataObject.getCustFirstName() method. If you are accessing a related DataObject, an accessor returns a strongly-typed DataObject rather than a regular DataObject. For example, `DataObject.get("Customers_Orders")` returns a DataObject, but `DataObject.getOrders()` returns an object of type Order.

To use static typing with the JDBC DMS, the metadata, a connection to the data source, and the DataGraph schema need to be provided to the JDBCMediatorFactory class `create` methods. In this case, the JDBC DMS metadata does not determine the shape of the DataGraph, but does give the DMS information about the backend data source and the way it maps to a DataGraph.

When using strongly typed DataObjects, it is important to make sure that the query matches the DataGraph schema. The query is not required to fill all of the data objects and properties in the schema, but a query cannot return data objects or properties that are not defined in the DataGraph schema. For example, a DataGraph schema might define `Customer` and `Order` DataObjects, but a query might only return Customer objects. Also, the Customer object might define properties for `ID`, `Name`, and `Address`, but the query might not return an address. In this case, the value of the address property is null, and the value is not updated in the database when the `applyChanges()` method is called. In this example, the query could not return a `Phone` property because it has not been defined as a property on the Customer object. When a query attempts this, the DMS returns an invalid metadata exception.

JDBC mediator supplied query:

An SDO client can supply the JDBC Data Mediator Service (DMS) with a SELECT statement to replace the statement that is generated from the DMS metadata.

When the SDO client instantiates a DMS, the DMS uses the defining metadata to generate a basic SELECT statement. Substituting that query gives you the ability to specify parameter markers; therefore you have more control over the client data that populates a dataGraph. Use a standard SQL SELECT string for a client-supplied query.

With both supplied queries and generated queries, UPDATE, INSERT, and DELETE statements are automatically generated for each DataObject. They are applied when the mediator commits the changes made to the DataGraph back to the database.

Parameter DataObjects for supplied queries

Clients can use a parameter DataObject to supply arguments to an SQL SELECT query. A parameter DataObject is a DataObject, but is not part of any DataGraph. It is constructed by the JDBC DMS when requested by the client. The ParameterDataObject for supplied queries is created based on the query given to the mediator. Every parameter in the query is given a name like *arg0, arg1, ..., argX*.

Because a parameter DataObject is a DataObject, you can set its properties using either the property name or an index value. The properties can be referenced by their argX name, or by the number associated with that parameter, 0, 1, ... , X. For example, your query is “SELECT CUSTFIRSTNAME WHERE CUSTSTATE = ? AND CUSTZIP = ?”. This supplied query contains two parameters. The first parameter corresponds with CUSTSTATE and can be set using the string “arg0” or the index 0. The second parameter corresponds with CUSTZIP and can be set using the string “arg1” or the index 1. Here is sample code of how they are set. This code assumes that you have already set up the metadata and mediator with the metadata and the aforementioned supplied query. Using the index value method, you code:

```
DataObject parameters = mediator.getParameterDataObject();
parameter.setString(0, "NY");
parameter.setInt(1, 12345);
DataObject graph = mediator.getGraph(parameters);
```

Using the property name method, you code:

```
DataObject parameters = mediator.getParameterDataObject();
parameters.setString("arg0", "NY");
parameters.setInt("arg1", 12345);
DataObject graph = mediator.getGraph(parameters);
```

The results are the same for both cases.

Limitations

The JDBC DMS generated SQL SELECT query is not fully supported on Oracle or Informix. This is because the mediator takes advantage of the ResultSetMetaData interface in JDBC 2.0 and requires it to be fully implemented. Oracle, Informix, DB2/390, and older supported versions of Sybase do not implement the ResultSetMetaData interface completely. The supplied select approach can still be used with these databases with one limitation: **column names in the Metadata must be unique across all tables.** An InvalidMetadataException occurs if the select statement returns a column with a name that appears multiple times in the metadata. For instance, if the Customer and the Order tables both contain a column named “ID”, this would be invalid and cause problems. The way to fix this is to change the name of at least one of the matching columns in the database to better distinguish the two columns from each other. For the Customer table, the column name could be changed to “CUSTID,” as it is in the examples. The Order column name could be changed to “ORDERID”. If you change the Customer column name, you do not have to change the Order column name, but for consistency it may be a good idea.

JDBC mediator generated query:

If you do not provide a structured query language (SQL) SELECT statement, then the data mediator service (DMS) generates one using the metadata provided at instance creation.

The internal query engine uses information in the metadata about tables, columns, relationships, filters, and order-bys to construct a query. As with the supplied queries, UPDATE, DELETE, and INSERT statements are automatically generated for each DataObject to be applied when the mediator commits the changes made to the DataGraph back to the database.

Filters

Filters define an SQL WHERE clause that might contain parameter markers. These are added to the DataGraph SELECT statement WHERE clause. Filters are used as is; they are not parsed or interpreted in any way so there is no error checking. If you use the wrong name, predicate, or function, it is not detected and the generated query is not valid. If a Filter WHERE clause contains parameter markers, then the corresponding parameter name and type are defined using Filter arguments. Parameter DataObjects fill in these parameters before the graph is retrieved. An example of the Filters and Parameter DataObjects for generated queries follows.

Limitation: Because of the tree-like nature of the DataGraph, any table at a branch appears in more than one subquery in the final union with the root table appearing in all paths. This means that it is not possible to filter on a table that appears in more than one path independent of all other paths. All filters defined on a particular table are joined by a boolean AND, and used everywhere that table appears.

Parameter DataObjects for generated queries

Clients use a Parameter DataObject to supply arguments that are applied to the filters provided in the DMS metadata. A Parameter DataObject is a DataObject, but is not part of any DataGraph. It is constructed by the JDBC DMS when requested by the client. The Parameter DataObject for generated queries is created based on the mediator's metadata. Every argument of every filter of every table is put into the Parameter DataObject. Unlike the supplied query Parameter DataObject, the parameters have the name assigned to them by the Filter arguments. The Parameter DataObject uses this name to map to the parameter to be filled in. The following sample code illustrates how a filter is created for a table in the mediator metadata. It also demonstrates the use of a Parameter DataObject to pass filter parameter values to a mediator instance. The sample assumes that the Customer table has already been defined:

```
// The factory is a MetadataFactory object
Filter filter = factory.createFilter();
filter.setPredicate("CUSTSTATE = ? AND CUSTZIP = ?");

FilterArgument arg0 = factory.createFilterArgument();
arg0.setName("customerState");
arg0.setType(Column.String);
queryInfo.getFilterArguments().add(arg0);

FilterArgument arg1 = factory.createFilterArgument();
arg1.setName("customerZipCode");
arg1.setType(Column.Integer);
queryInfo.getFilterArguments().add(arg1);

// custTable is the Customer Table object
custTable.setFilter(filter);


..... // setting up mediator


DataObject parameters = mediator.getParameterDataObject();
```

```
// Notice the first parameter is the name given to the
// argument by the FilterArgument.
parameter.setString("customerState", "NY");
parameter.setInt("customerZipCode", 12345);
DataObject graph = mediator.getGraph(parameters);
```

Order-by

Ordering of query results is specified using OrderBy objects that identify a column from a table to sort the results. This ordering can be either ascending or descending. The OrderBy objects are part of the metadata and are automatically applied to generated queries. An example of this for a customer table results to be sorted by first names is as follows:

```
// This example assumes that the custTable, a table in
// the metadata, and factory, the MetaDataFactory
// object, have already been created.
Column firstName = ((TableImpl)custTable).getColumn("CUSTFIRSTNAME");
OrderBy orderBy = factory.createOrderBy();
orderBy.setColumn(firstName);
orderBy.setAscending(true);
metadata.getOrderBys().add(orderBy);
```

Limitation: Even though Order-bys are defined on each table in the metadata, the RDBMS model requires them to be applied to the final query. This has many implications. For example, you cannot order a table and then use that in a join to another table and propagate the ordering in the first table. Because a result set is a union of all the tables in the DataGraph, the nature of the single result set requires that it be padded with nulls, which can affect the order-bys, particularly in the non-root tables. This can give unexpected results.

External Tables

An external table is a table defined in the metadata that is not needed in the DataGraph returned by the JDBC DMS. This might be appropriate when you want to filter the result set based on data from a table but that table's data is not needed in the result set. An example of this with the Customers and Orders relationship would be to filter the results to return all customers who ordered items with an order date of the first of the year. In this case, you do not want any order information returned, but you do need to filter on the order information. Making the Orders table external excludes the orders information from the DataGraph and therefore reduces the DataGraph's size, improving efficiency. To designate a table as external, you call the setExternal(true) method from a table object in the JDBC DMS metadata. If the client tries to access an external table from the DataGraph, an illegal argument exception occurs.

Limitation: Many RDBMSs require that an orderby column appear in the final result set; the columns from an external table cannot in general be used to order a result set. Order-bys are actually applied to the result set (the word “set” is key here), and not to intermediate query results.

General limitations of generated queries

In understanding the limitations of the query generation feature in the JDBC DMS, there are two things to keep in mind. The first is that the DataGraph imposes a model that is a directed, connected graph with no cycles (that is, a model that is a tree) on a relational model that is a non-directed, potentially disconnected graph with cycles. `Directed` means that the developer chooses the orientation of the graph by picking a root table. `Connected` means that all tables that are a member of the DataGraph are reachable from the root. Any tables that are not reachable from the root cannot be included in the DataGraph. In order for a table to be reachable from the root, there must be at least one foreign key relationship defined between each pair of tables in the DataGraph. `No cycles` means that there is only one foreign key relationship between a pair of tables in the DataGraph. The tree nature of the DataGraph determines how the queries are built, and what data is returned from a query.

The second item to keep in mind is the following high level description of how query generation produces read queries for a DataGraph:

1. The JDBC DMS creates a single result set (that is, a DataGraph) whether the DataGraph is composed from a single table or from multiple tables.
2. Each path through the foreign key relationships in DMS Metadata from root to leaves represents a separate path. The data for that path is retrieved by using joins across the foreign keys defined between the tables in the path. The joins are by default inner joins.
3. All the paths in a DataGraph are unioned together in order to create a single result set by the query that is generated by the mediator, and are thus treated independently of one another.
4. Any user-defined filtering is done first on the tables. Then the result is joined to the rest of the path.
5. Relational databases generally require order-bys to be applied to the entire final result set and not on intermediate results.

JDBC mediator performance considerations and limitations:

Use these tips to help you determine if a JDBC Data Mediator Service suits the requirements of your application serving environment.

Miscellaneous database limitations

- Sybase before Version 12.5.1 does not support in-line queries in the “from” clause, and therefore does not support multiple table DataGraphs with filters. To use the Service Data Object in WebSphere Application Server use Sybase Version 12.5.1.
- The Informix Dynamic Server does not support sub-selects, which are needed for multiple table graphs. Use Informix Extended Parallel Server.
- Oracle 8i does not support the ANSI join syntax. The mediator in multiple table cases requires Oracle 9i or 10g.

General performance recommendations

- Evaluate if your target projects are well suited to these technologies. In general, projects that are read-intensive and require disconnected data are good candidates.
- Limit the number of tables in the metadata. One or two is best because relationships, with respect to filters, become ambiguous when graphs have many branches.
- Work with small data sets as often as possible to avoid consuming excessive amounts of memory within your applications. You can limit the amount of data returned to the SDO by specifying filters in the metadata objects or by using paging.
- For web applications, if the DataGraph is not too large and is to be reused later, store it in the user session.

JDBC mediator transactions:

You can specify that the JDBC mediator either act as transaction manager, or refrain from such activities in the case of external transaction management (performed by the SDO client).

Mediator managed transactions

A JDBC connection is wrapped in a connection wrapper and passed to the Data Mediator Service (DMS) during the instance creation. The ConnectionWrapper object contains the connection that is used by the JDBC DMS and indicates whether the mediator manages the current transaction. When the JDBC DMS manages the transaction, it performs commit and rollback operations as required. However, the DMS does not perform any transaction management activities if the wrapped connection is currently engaged in another transaction.

Using the createConnectionWrapper method for active transaction management is the general practice.

Non-mediator managed transactions

When a *passive* connection wrapper is passed to the DMS, the DMS takes no managerial action; a passive wrapper is generally intended for an existing transaction that is under external management. Commit or rollback operations are not performed by the connection wrapper in this case.

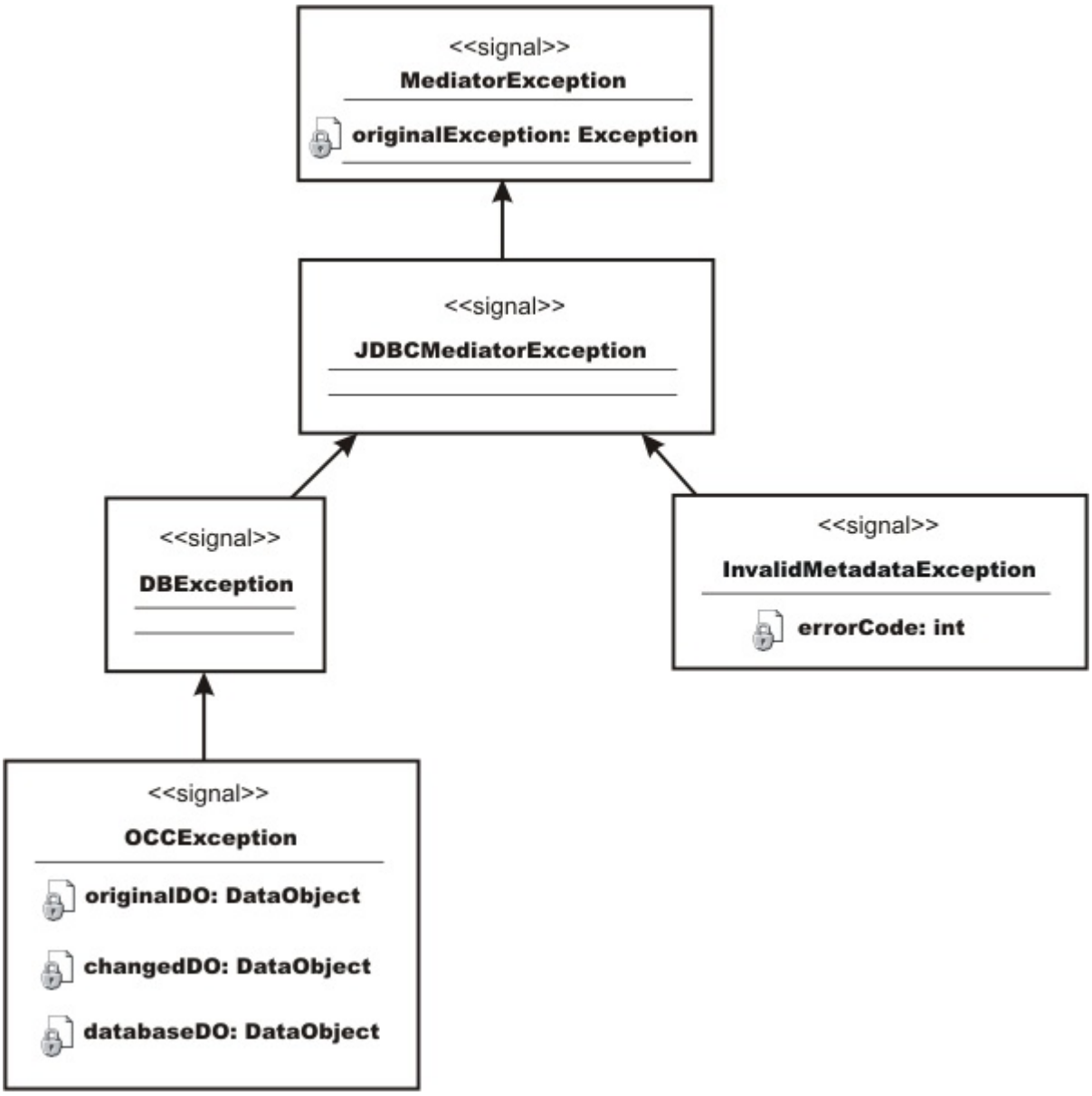
Use the createPassiveConnectionWrapper method.

Protection against referential integrity (RI) violations

The JDBC Data Mediator Service safeguards data transactions from incurring RI violations and other database logic violations. When the JDBC DMS applies the updates of a data graph to a back end, it automatically orders the change operations so that they do not violate database RI policy. Similarly, the DMS filters counter operations (such as INSERT and DELETE) so that opposing client requests can perform updates in a logical order. The client deletes one object, and then creates an entirely separate object with the same primary key. The DMS transforms these two operations into an update operation that modifies the existing database object.

JDBC mediator exceptions:

JDBC mediator exceptions either surface errors reported by the database, or indicate use of non-valid metadata in the attempt to instantiate the DMS.



The `Mediator` exception is the root exception of all the data mediator services, and the `JDBCMediator` exception is the root exception for the JDBC DMS in particular.

The `DB` exception occurs when an error is reported by the database. This can occur several ways:
- when the connection being used has the AutoCommit property set to *true*, but the JDBC DMS is controlling the transaction and needs it to be set to false
- when an unsupported database is trying to be used
- when other backend database errors occur during commit or rollback.

An optimistic concurrency control (OCC) exception occurs when the applyChanges() operation results in an data collision. When this occurs, the exception contains the original row values, current row values, and the attempted row values. These values are used to help recover from the error.

An `InvalidMetadata` exception occurs for invalid metadata supplied to the JDBC DMS upon creation. This can happen when a query requires tables or columns that are not defined in the metadata, or when there are identical column names for different tables for the Oracle, Informix, and older supported versions of Sybase databases.

Example: Forcing OCC data collisions and JDBC mediator exceptions, The following example forces a collision to demonstrate detection and shows the exception that occurs as a result.

```
// This example assumes that a mediator has already
// been created and the first name in the list is Sam.
// It also assumes that the Customer table has an OCC
// column and the metadata has set this column to be
// the collision column.

DataObject graph1 = mediator.getGraph();
DataObject graph2 = mediator.getGraph();

DataObject customer1 = (DataObject)graph1.getList("CUSTOMER").get(0);
customer1.set("CUSTFIRSTNAME", "Bubba");

DataObject customer2 = (DataObject)graph2.getList("CUSTOMER").get(0);
customer2.set("BOWLERFIRSTNAME", "Slim");

mediator.applyChanges(graph2);

try
{
 mediator.applyChanges(graph1);
}
catch (OCCException e)
{
// Since graph1 was obtained before graph2 and
// graph2 has already been submitted, trying to
// apply the same changes to graph1 causes
// this OCC Exception.

assertEquals("Sam", e.getOriginalDO(). getString("CUSTFIRSTNAME"));
assertEquals("Bubba", e.getChangedDO(). getString("CUSTFIRSTNAME"));
assertEquals("Slim", e.getDatabaseDO(). getString("CUSTFIRSTNAME"));
}
```

Defining optimistic concurrency control for JDBC Mediator:

Implement an optimistic concurrency control (OCC) strategy for the JDBC DMS to diagnose transaction problems that are caused by update collisions.

About this task

An *update collision* occurs when client data that populates a data graph is changed in the database before the data graph can submit the modifications of the client. If you configure the JDBC DMS for OCC, the DMS issues an OCC-specific exception when such a data collision happens. The OCC exception contains collision details such as the original row values, current row values, and the attempted row values. The client application uses these values to determine how to recover from the collision. For example, the application can reread the data and restart the transaction.

Be aware, however, that when one exception occurs, there is no way of knowing whether more exceptions exist deeper in the data graph schema and therefore are not displayed.

To activate OCC for the data mediator service, you must incorporate OCC columns into your database tables.

Add an *OCC Integer* column to a given table, and specify that this column is to be used for OCC in the metadata. The defined OCC collision column is reserved for the exclusive use of the mediator. If there is no OCC column defined for a table, the DMS does not monitor and notify you of update collisions. The following generic code segments create this setup.

Procedure

1. Create the OCC column
   ```
   Column collisionColumn = table.addIntegerColumn("OCC_COUNT");
   ```
2. Ensure that it does not allow null values
   ```
   collisionColumn.setNullable(false);
   ```
3. Designate the column as the table collision column
   ```
   table.setCollisionColumn(collisionColumn);
   ```
   For a fully-fledged code example that forces a collision to demonstrate the OCC exception, see the topic Example: Forcing OCC data collisions and JDBC mediator exceptions.

JDBC mediator integration with presentation layer:

The JDBC Data Mediator Service (DMS) can be used in conjunction with web application presentation layer technologies such as JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces (JSF).

This discussion assumes a general understanding of both of the JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces (JSF) technologies. In particular for JSF, the UIData component and the general file structure of a JSF dynamic web application should be known. For a brief overview of both JSF and JSTL refer to the topics under Service Data Objects.

The JDBC DMS and JSTL work well together because the JSTL access code is equivalent to the code necessary to access attributes and lists inside of a DataObject. For example, in relation to a root Customer DataObject, the JSTL expression:

```
${rootDO.CUSTOMER[index].CUSTNAME}
```

is equivalent to the Java code for a DataObject of:

```
rootDO.getList("CUSTOMER").get(index).get("CUSTNAME")
```

The reason for this is the dot notation in the JSTL expression language correlates to a *getter()* method in Java code, and the bracket notation allows you to access elements inside a list.

The JDBC DMS and JSF fit well together because the DataGraph produced by the JDBC DMS is able to populate a JSF UIData component without having to be transformed. The UIData component uses a *dataTable* tag that takes a list as its input to populate the table. This works out well with the DataGraph because all you need to pass into the dataTable is the root list of the DataGraph. The most common way to lay out the DataGraph in the dataTable is to display each attribute of the DataObject from the list retrieved from the root in its own column, and to embed each additional relationship to the DataObject in a new dataTable contained within the parent DataObject's row. Using this method instead of a traditional ResultSet table eliminates duplicate information and makes it easier to see the separation of the parent object's children. An example of how the Customer and Order scenario is laid out in a dataTable is shown in the topic Example: Using JavaServer Faces and JDBC Mediator dataTables.

Example: Using JavaServer Faces and JDBC Mediator dataTables.

This example shows code that would be located inside of a Faces JSP page. It demonstrates how to use JavaServer Faces and JDBC Mediator dataTables in an application.

It contains the UIData component dataTable tag with all of the customer's information, along with their orders. Each Customer attribute has its own column. The Customer Orders are embedded in another

dataTable containing each of the Order attributes in separate columns. This embedded dataTable of Orders is like any other Customer attribute, having its own column inside each Customer row.

```
<h:dataTable id="table1" value="">{pc_Customers.customer}" var=
"varcustomer" styleClass="dataTable">

  <h:column id="column1">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Customerid" id=
        "text1"></h:outputText>
    </f:facet>
    <h:outputText id="text2" value="">{varcustomer.CUSTOMERID}"
    styleClass="outputText">
     <f:convertNumber />
    </h:outputText>
    </h:column>

  <h:column id="column2">
    <f:facet name="header">
 <h:outputText styleClass="outputText" value="Custfirstname"
     id="text3"></h:outputText>
   </f:facet>
   <h:outputText id="text4" value="">{varcustomer.CUSTFIRSTNAME}"
     styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column3">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custlastname"
        id="text5"></h:outputText>
    </f:facet>
    <h:outputText id="text6" value="">{varcustomer.CUSTLASTNAME}"
      styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column4">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custstreetaddress"
        id="text7"></h:outputText>
    </f:facet>
    <h:outputText id="text8" value="">{varcustomer.CUSTSTREETADDRESS}"
      styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column5">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custcity" id="text9">
        </h:outputText>
    </f:facet>
    <h:outputText id="text10" value="">{varcustomer.CUSTCITY}"
       styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column6">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custstate" id=
         "text11"></h:outputText>
    </f:facet>
    <h:outputText id="text12" value="">{varcustomer.CUSTSTATE}"
      styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column7">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custzipcode"
         id="text13"></h:outputText>
    </f:facet>
```

```
    <h:outputText id="text14" value="">{varcustomer.CUSTZIPCODE}"
       styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column8">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custareacode"
         id="text15"></h:outputText>
    </f:facet>
    <h:outputText id="text16" value="">{varcustomer.CUSTAREACODE}"
       styleClass="outputText">
    <f:convertNumber />
    </h:outputText>
  </h:column>

  <h:column id="column9">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Custphonenumber"
         id="text17"></h:outputText>
    </f:facet>
    <h:outputText id="text18" value="">{varcustomer.CUSTPHONENUMBER}"
      styleClass="outputText">
    </h:outputText>
  </h:column>

  <h:column id="column10">
    <f:facet name="header">
      <h:outputText styleClass="outputText" value="Customers_orders"
         id="text19"></h:outputText>
    </f:facet>

    <h:dataTable id="table2" value="">{varcustomer.CUSTOMERS_ORDERS}"
      var="varCUSTOMERS_ORDERS" styleClass="dataTable">

     <h:column id="column11">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Ordernumber"
        id="text20"></h:outputText>
   </f:facet>
   <h:outputText id="text21"
      value="">{varCUSTOMERS_ORDERS.ORDERNUMBER}"
      styleClass="outputText">
     <f:convertNumber />
   </h:outputText>
 </h:column>

 <h:column id="column12">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Orderdate"
        id="text22"></h:outputText>
   </f:facet>
  <h:outputText id="text23" value="">{varCUSTOMERS_ORDERS.ORDERDATE}"
     styleClass="outputText">
     <f:convertDateTime />
   </h:outputText>
 </h:column>

 <h:column id="column13">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Shipdate"
        id="text24"></h:outputText>
   </f:facet>
   <h:outputText id="text25"
     value="">{varCUSTOMERS_ORDERS.SHIPDATE}"
     styleClass="outputText">
     <f:convertDateTime />
   </h:outputText>
 </h:column>

 <h:column id="column14">
   <f:facet name="header">
```

```
    <h:outputText styleClass="outputText" value="Customerid"
      id="text26"></h:outputText>
   </f:facet>
   <h:outputText id="text27"
       value=">{varCUSTOMERS_ORDERS.CUSTOMERID}" styleClass="outputText">
     <f:convertNumber />
   </h:outputText>
 </h:column>

 <h:column id="column15">
   <f:facet name="header">
     <h:outputText styleClass="outputText" value="Employeeid"
       id="text28"></h:outputText>
   </f:facet>
   <h:outputText id="text29"
       value=">{varCUSTOMERS_ORDERS.EMPLOYEEID}" styleClass="outputText">
     <f:convertNumber />
   </h:outputText>
 </h:column>

    </h:dataTable>
  </h:column>
</h:dataTable>
```

JDBC mediator paging:

Paging can be useful for moving through large data sets because it can limit the amount of data pulled into memory at any given time. The JDBC DMS API provides two interfaces that implement paging.

If the metadata provided to the data mediator service (DMS) defines customers and the page size is set to ten, then the first page is a DataGraph containing the first ten customer DataObjects. The next page is another DataGraph with the next ten Customers, and so forth.

One thing to note is that the JDBC DMS provides paging at the root of the graph. That is, there is no restriction on the number of related DataObjects returned. For example, if the metadata provided to the DMS defines customers and related orders, it is the customers that are paged. If the page size is set to ten, then the first page is a graph with the first 10 customers and all related orders for each customer.

There are two interfaces provided by the DMS that you can take advantage of, the **Pager** and the **CountingPager**. The Pager interface provides a cursor-like *next()* method capability. The next() function returns a graph representing the next page of data from the entire data set specified by the mediator metadata. There is also a *previous()* function available with the same capabilities, only going backward. The CountingPager interface enables you to retrieve a specific page number. The following example illustrates paging through a large set of customer instances using a CountingPager interface with a maximum of 5 DataObjects from the root table per page.

```
CountingPager pager = PagerFactory.soleInstance.createCountingPager(5);
int count = pager.pageCount(mediator);
for (int i = 1, i <= count, i++) {
DataObject graph = pager.page(i, mediator);
// Iterate through all returned customers in the
// current page.
Iterator iter = graph.getList("CUSTOMER").iterator();
 while (iter.hasNext()) {
  DataObject cust = (DataObject) iter.next();
System.out.println(cust.getString("CUSTFIRS NAME"));
 }
}
```

If you try to move before the first page or after the last available page, a JDBC mediator exception occurs.

JDBC mediator serialization:

The DataGraph produced by the JDBC DMS can be serialized and written out to a file, or sent across a network.

The following example illustrates serialization and de-serialization of a graph:

```
// This example assumes the creation of the Customer
// metadata and the JDBC DMS.

DataObject object = mediator.getGraph();
DataGraph origGraph = object.getDataGraph();

FileOutputStream out = new FileOutputStream("test.datagraph");
ObjectOutputStream oos = new ObjectOutputStream(out);
oos.writeObject(origGraph);
out.close();

FileInputStream in = new FileInputStream("test.datagraph");
ObjectInputStream oin = new ObjectInputStream(in);
DataGraph graph = (DataGraph) oin.readObject();
DataObject obj = (DataObject) graph.getRootObject();

// Now, the DataObject retrieved from the input stream
// obj is equal to the original variable object put
// through the output stream.
```

Enterprise JavaBeans Data Mediator Service:

The Enterprise JavaBeans (EJB) Data Mediator Service (DMS) is the Service Data Objects (SDO) Java interface that, given a request in the form of EJB queries, returns data as a DataGraph containing DataObjects of various types.

This is different from a typical EJB finder or ejbSelect method, which also takes an EJB query but returns a collection of EJB objects that are all of the same type or a collection of container managed persistence (CMP) values.

The EJB DMS enables you to specify an EJB query that returns a data graph (the DataGraph) of data objects (DataObjects). The query can be expressed as a compound EJB query that is contained in a string array of EJB query statements. One advantage of using a DataGraph is that much of the code written in an EJB facade session bean that creates, populates, and updates copy helper objects can be replaced with a DataGraph and a DMS.

Important: The EJB DMS has support for EJB2.x container managed persistence (CMP) entity beans only. It does not support EJB 3.x modules.

You can obtain a DataGraph using the getGraph call, either from EJB instances cached in the container, or the query request can be compiled into SQL and executed directly against the data source.

Updated DataObjects can be written back to the data store by using the applyChanges method in one of two ways. The updates can be translated into SQL and applied directly to the data store or can be written back through EJB accessor methods. Writing back directly to the data store can improve performance because it avoids EJB activation. However, if business logic or EJB container function is required by the application, writing back through EJB is the preferred approach. When writing back through EJB, you can specify a user-defined MediatorAdapter method to ensure customized handling of changed DataObjects. This customization can include application-specific optimistic concurrency control, invoking business methods on the EJB to perform updates, update of computed values in the DataObject, and calling application-specific create methods on EJBHome.

Update processing is not dependent on how the DataGraph was originally retrieved. In other words, it is possible to retrieve a DataGraph directly from the data source, but have the deferred updates applied through the enterprise bean or the other way around.

Regardless of which update approach you use, an optimistic concurrency control algorithm is used. Fields designated as consistency fields are read during the update to ensure that the current value is equal to the old value DataObject field.

Runtime processing

An EJB mediator request is a compound EJB query, which consists of an ordered list of regular EJB queries. Each query in the compound query defines an SDO. The SELECT clause of the query specifies the CMP fields or expressions to return in the DataObject. The WHERE clause specifies the filtering conditions. The first query in the list is considered to be the ROOT node in the DataGraph. The FROM clause of a query, other than the first, specifies an EJB relationship that is used to create references between DataObjects. More details about how the DataGraph schema is derived from the query can be found in the topic DataGraph schema.

EJB data mediator service programming considerations:

When you begin writing your applications to take advantage of the Enterprise JavaBeans (EJB) data mediator service (DMS) provided in the product, consider the following items.

EJB programming model

Only a subset of the EJB programming model is supported by the EJB data mediator service.

- When using EJB collection parameters to retrieve data from EJB instances, or when using applyChanges to update EJB instances:
  - The EJB DMS uses local interfaces for enterprise beans. `Getter` and `setter` calls for container-managed persistence (CMP) fields must be promoted to the local interface, as well as any EJB methods used in query expressions.
  - For the mediator to create an EJB, there must be a create method using the primary key class as the only argument method defined on the EJB home. If no such method exists, you must supply an adapter that handles the create operation. Also, the EJBLocalHome interface defined for the EJB must include (in addition to the create method) the following method:

    ```
    findByPrimaryKey(<key class>)
    remove (java.lang.Object)
    create (<key class>)
    ```
- When invoking the applyChanges method directly to the database, the following occur:
  - you bypass container update. You should force a refresh as soon as possible by transaction termination and using appropriate container cache options.
  - you bypass EJB container-managed relationship (CMR) maintenance. You must rely on database RI to maintain those relationships not retrieved into the DataGraph.
- CMP fields must be the allowed types. See “EJB mediator query syntax” on page 146 for a list of those types.
- CMP fields of user-defined types that use EJB converters/composer are not supported.

The following table shows limitations in the EJB programming model that are not supported by the EJB DMS.

Table 10. EJB programming model limitations with EJB DMS. The following table shows limitations in the EJB programming model that are not supported by the EJB DMS.

	retrieve direct from db	**retrieve from EJB Container**	**update direct to db**	**update through EJB**
EJB persistence inheritance	No	No	No	No
EJB cmp field with converter	No	Yes	No	Yes

Transactional

- All mediator calls, including create, must be done within a transaction scope – either a user transaction or a container transaction. The various mediator calls including, create, getGraph, and applyChanges, do not have to be called within the same transaction. In fact, most often the calls are done in separate transactions.

Access intent

- When the mediator query references an EJB using its abstract schema name (ASN), data is retrieved directly from the database. The access intent and isolation level used on the data source connection is the access intent specified in the application profile for EJB dynamic query access intent. It is recommended that you define an optimistic access intent for your application because a DataGraph is intended to be used in a disconnected programming model.
- When the mediator is retrieving data using an EJB collection, the access intent specified in the application profile is used if the EJB requires activation.
- During applyChanges, optimistic concurrency control is used to verify certain fields in the DataObject before applying changes to the database. Updates are typically processed under a different transaction from the retrieval. Therefore, to avoid lost updates it is necessary to verify that another transaction has not updated the data. When defining the EJB to RDB mapping you can specify one or more EJB fields as optimistic Predicates. The fields are used for verification by comparing the current database value to the old value from the DataGraph change log. If the verification succeeds, then the current value of the fields is written to the database. If the comparison returns false and the update fails, an exception occurs. All of this is accomplished in a single update statement with extra predicates added, such as in the following example. The optimisticPredicate field is `myColumn1`.

  ```
  update  myTable set  myColumn1 = "new value1", myColumn2="new value2"
  where myKey= "key value" and myColumn1 ="old value1"
  ```

- When applyChanges is done through the EJB container, the current values of the enterprise beans are compared with the old values of the optimistic predicates fields. If the values are unequal an exception occurs.
- Provided that you have defined one or more EJB fields as optimisticPredicates, then for the SDO to be updateable, at least one of the optmisticPredicate fields must be retrieved into the data object. Otherwise, applyChanges returns an exception. The field should be updated either by the caller or a database trigger – the mediator does not automatically increment or set the field.
- Not all fields are verified, only those fields marked as optimisticPredicate in the EJB-RDB mapping.
- Note that the EJB mapping tool allows for the possibility of no optimisticPredicate fields. In this case the mediator will perform updates without any verification.
- Creation and deletion operations do not make use of the optimistic predicate fields.
- When applying changes through EJB instances, the EJB might have to be activated first. In this case, the appropriate access intent associated with the EJB methods apply. It is recommended that you run applyChanges in a profile that has pessimistic access intent, otherwise the optimistic concurrency logic is invoked twice – once when copying data object values to the EJB, and a second time when the persistence manager compares the old values of the EJB field values against the database record.
- The access intent used by the mediator when retrieving directly from the database is the default access intent defined for the EJB named in the first query statement.

Best practices

- You can call getGraph on one mediator instance, update the returned DataGraph, and then call applyChanges on a different mediator instance. However, while they do not need the same mediator instance, they do need the same `query shape`. The query shape is the number and order of query statements, the fields and relationships specified in the SELECT and FROM clauses, and so on.
- Avoid repeated calls to createMediator if possible. Use parameterized queries and use getGraph to pass in different parameter values.

EJB data mediator service data retrieval:

An Enterprise JavaBeans (EJB) mediator request is a compound EJB query. You can obtain a DataGraph using the *getGraph* call.

Directly from the data source

To retrieve data directly from the data source, specify your first EJB query to reference the Abstract Schema Name (ASN) of the enterprise bean.

From the EJB container

To retrieve data through the EJB container, specify your first query to use an input parameter in the FROM clause referring to the EJB collection desired.

You should use this method when there is high likelihood that your EJB instances will be cached in the container. This way you avoid container flush and then read from the database to retrieve data.

For an example, see the topic Using the EJB data mediator service for data access.

EJB data mediator service data update:

An Enterprise JavaBeans (EJB) mediator request is a compound EJB query. You can write an updated DataGraph back to the data source by using the applyChanges method.

The update can be applied directly to the data source or through EJB instances.

When applying changes through EJB instances an optional adapter class can be specified on the applyChanges method. Each changed data object is first passed to the adapter applyChange method. The adapter can process the change itself and return **true**, or have the EJB Mediator process the change by returning **false**.

The adapter can be used to customize the optimistic concurrency (OCC) logic, or process changes to read only DataGraph attributes, or process changes that require business logic.

There are two forms of the applyChanges method. The first, *applyChanges(DataObject)* takes the updated DataGraph and runs structured query language (SQL) insert, update, and delete statements directly against the database, bypassing the EJB container. The second form, *applyChanges(DataObject, MediatorAdapter)* processes updates using EJB instances and accessors. A null value for the MediatorAdapter is supported.

When to use an adapter with applyChanges

- Use when there are create methods other than create(PrimaryKey)
- Use when business methods must be called instead of container-managed persistence (CMP) *setter* methods
- Use when special optimistic caching logic is needed

How the adapter works

Three passes are made over the DataGraph log, passing changed DataObject to the adapter:

1. New DataObjects are passed. The adapter can create the object and set the CMP fields. Container-managed relationships (CMR) that reference enterprise beans not yet created are deferred until pass 2.
2. New and updated DataObjects are passed. CMRs deferred from pass 1 can be set at this time.
3. Deleted DataObjects are passed.

EJB mediator query syntax:

When you begin writing your applications to take advantage of the Enterprise JavaBeans (EJB) data mediator service (DMS) provided in the product, consider the following items.

- The EJB DMS takes as an input argument a compound EJB query which consists of an array containing EJB query language (QL) statements and an optional XREL command. The XREL command is a list of EJB relationships and must appear last in the array.
- Each EJB QL query returns data in the form of a Service DataObjects (SDO) instance. All of the SDO instances are merged into a DataGraph. The SELECT clause of each query specifies the container-managed persistence (CMP) fields or expressions to return in the SDO. The WHERE clause specifies the filtering conditions and you can define an ORDER BY clause. If two or more SELECTs return the same SDO type, each SELECT must project the same CMP fields and expressions. For updatability, the primary key fields of the EJB must be projected. JOINs, UNIONs, and aggregation are not supported except in subqueries.
- A query in the array can refer to a prior query in the FROM clause by using the identification variable defined in the prior query and a relationship name. This relationship can be single or collection valued.
- Relationships are constructed between data object instances in the graph when a relationship is used in either the FROM clause or in the XREL command.
- Collection valued input arguments are supported in FROM clause If ?1 refers to a collection of Dept EJBs then the following query is valid for the mediator. The cast syntax is required to tell the query compiler the collection element type.

  ```
  select d.deptno from (Dept) ?1 as d
  ```

- The collection input argument is useful when it is desired to build a DataGraph from EJB instances that are cached in the EJB Container or persistence manager data cache.
- The SELECT clause can specify a list of CMP fields to retrieve (the wildcard * notation can be used to retrieve all CMP fields) or valid EJB query language expressions. CMP fields and expressions must be one of the following types:
  - Primitive types: boolean, byte, short, integer, long, float, double, char
  - Object wrapper types for the primitive types
  - Java.lang.String
  - Java.math.BigDecimal
  - java.math.BigInteger
  - byte [ ]
  - Java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - java.util.Date
  - java.util.Calendar
- All primary key CMP fields must be retrieved in order for the Service Data Objects (SDO) to be updateable; otherwise, applyChanges returns an exception.
- SDO attributes that come from EJB query language expressions such as *e.salary + e.bonus AS TOTAL_PAY* cannot be updated. If you try to make an update, applyChanges returns a QueryException.
- Aggregate expressions such as *SUM(e.salary)* are not allowed even though they are part of the EJB query language. Aggregate expressions can be used in subselects in the WHERE clause.

XREL keyword:

The XREL keyword is used to build relationships independent of how the data was retrieved. XREL is valid only in Enterprise JavaBeans (EJB) Mediator queries.

XREL does not retrieve additional data, it only builds relationships from data already retrieved by the select statements. The relationships can be one-to-one, one-to-many, many-to-one, or many-to-many. The

relationships can be unidirectional or bidirectional. If you specify a bidirectional relationship in an XREL, the inverse relationship is also established in addition to the specified relationship.

```
xrel := XREL identification_variable . { single_valued_cmr_field | collection_valued_cmr_field }
            [ , identification_variable . { single_valued_cmr_field | collection_valued_cmr_field } ]*
```

Examples: XREL keyword

This example retrieves all employees and all departments, and establishes the *emps* and *mgr* relationships.

```
select e.name from EmpBean e
  select  d.name from DeptBean d
  xrel  d.emps, d.mgr
```

Notice that the employees are retrieved through d.emps relationship, xrel d.mgr is to establish the *mgr* relationship for those employees who are also a manager.

```
 select  d.name from DeptBean d
   select e.name from in(d.emps) e
   xrel  d.mgr
```

DataGraph schema:

DataGraph schema created by the EJB mediator

The schema created by the mediator for a query consists of an Eclass for each query statement. The name of the Eclass is the Abstract Schema Name (ASN) of the Enterprise JavaBeans (EJB). The Eattributes of the Eclass correspond to the container-managed persistence (CMP) fields or expressions returned by the query statement.

For static DataObjects, the Eclass name can be different provided that the Map argument is used on the createMediator call.

Each EJB relationship specified in the FROM or XREL clause adds an Ereference into the schema. EJB relationships can be unidirectional or bidirectional. However, all Ereferences are defined as bidirectional as this is needed to efficiently navigate the DataGraph on update. An inverse relationship name is generated in the case of a unidirectional EJB relationship. A generated name is of the format <ASName_source><ASName_target>. For example, if the ASNames are EmpBean and DeptBean, and the unidirectional relationship is *dept* going from EmpBean to DeptBean, the generated inverse name is **DeptBeanEmpBean**.

If no ECLass argument is used on createMediator, then the mediator creates a DataGraph schema with the following characteristics:

- the DataObject Eclass names are the corresponding EJB Abstract Schema Names (ASN)
- the DataObject attributes names and types are the expression names and types in the query SELECT clauses
- the DataObject reference names and types come from the EJB relationships referenced in the FROM clauses.

A *dummy* DataObject with the Eclass name of *DataGraphRoot* is also created and has containment reference to all the DataObjects. The reference is multivalued, using the EJB ASN name.

```
DataObject root =  m.getGraph( parms );
root.getType().getName();   // this would return the string "DataGraphRoot"

List depts = (List) root.get("DeptBean");
// the list of all DeptBean SDOs in the DataGraph

List emps = (List) root.get("EmpBean");
// the list of all EmpBean SDOs in the DataGraph
```

DataGraph containment patterns

References between Service Data Objects (SDO) can be defined as containment references, in which case when an SDO is deleted the delete is cascaded to all of the contained SDO. Also, when the DataGraph is serialized as an XML document, the contained SDO are nested within the parent SDO. Noncontained references are expressed as path expressions in the XML document.

Containment must be defined in the DataGraph schema. When the mediator defines the schema, the root SDO (named *DataGraphRoot*) contains all other SDO. EJB relationships are defined as noncontained SDO references.

When the caller defines the DataGraph schema, there are three patterns.

ROOT_CONTAINS_ALL
: In this pattern there is a dummy SDO that is the root. It is a dummy in the sense that it does not correspond to any EJB. Its purpose is to contain all other SDOs. If the mediator generates the graph schema, the dummy root has a class name of DataGraphRoot and it will have containing references whose names are the EJB ASN names. If the caller uses static schema, the root can have any name. The Eclass of the root is passed on the createMediator call.

ROOT_CONTAINS_SOME
: This pattern is applicable only for static schema. There is still a dummy SDO that is the graph root. Other SDO must either be contained by the Ereference that corresponds to the EJB relationship used in the query statement or the SDO must be contained by the dummy root.

NO_DUMMY ROOT
: This pattern is applicable only for static schema. There is no dummy root. The root SDO corresponds to the first query statement which must return only a single instance. Non-root SDOs must be contained by the Ereference corresponding to the EJB relationship used in the query statement.

Using the Java Database Connectivity data mediator service for data access

The following steps demonstrate how to create the metadata for a Java Database Connectivity (JDBC) data mediator service (DMS), as well as how to instantiate the DMS dataGraph.

Procedure

1. Create the metadata factory. This can be used for creating metadata, tables, columns, filters, filter arguments, database constraints, keys, order-by objects, and relationships.

   ```
   MetadataFactory factory = MetadataFactory.eINSTANCE;
   Metadata metadata = factory.createMetadata();
   ```

2. Create the table for the metadata. You can do this two ways. Either the metadata factory can create the table and then the table can add itself to the already created metadata, or the metadata can add a new table in which case a new table is created. Because it involves fewer steps, this example uses the second option to create a table called CUSTOMER.

   ```
   Table custTable = metadata.addTable("CUSTOMER");
   ```

3. Set the root table for the metadata. Again, you can do this in two ways. Either the table can declare itself to be the root or the metadata can set its own root table. For the first option, code:

   ```
   custTable.beRoot();
   ```

   If you want to use the second option, you code:

   ```
   metadata.setRootTable(custTable)
   ```

4. Set up the columns in the table. The example table is called CUSTOMER. Each column is created using its type. The column types in the metadata can only be the types supported by the JDBC driver being used. If you have questions on which types the JDBC driver being used supports, consult the JDBC driver documentation.

```
Column custID = custTable.addIntegerColumn("CUSTID");
custID.setNullable(false);
```

This example creates a column object for this column, but does not for the remainder. The reason is because this column is the primary key, and is used to set the table's primary key after the rest of the columns are added. A primary key cannot be null; therefore custID.setNullable(false) prohibits this from happening. Adding the rest of the columns:

```
custTable.addStringColumn("CUSTFIRSTNAME");
custTable.addStringColumn("CUSTLASTNAME");
custTable.addStringColumn("CUSTSTREETADDRESS");
custTable.addStringColumn("CUSTCITY");
custTable.addStringColumn("CUSTSTATE");
custTable.addStringColumn("CUSTZIPCODE");
custTable.addIntegerColumn("CUSTAREACODE");
custTable.addStringColumn("CUSTPHONENUMBER");

custTable.setPrimaryKey(custID);
```

5. Create other tables as needed. For this example, create the Orders table. Each order is made by one Customer.

```
Table orderTable = metadata.addTable("ORDER");

Column orderNumber = orderTable.addIntegerColumn("ORDERNUMBER");
orderNumber.setNullable(false);

orderTable.addDateColumn("ORDERDATE");
orderTable.addDateColumn("SHIPDATE");
Column custFKColumn = orderTable.addIntegerColumn("CUSTOMERID");

orderTable.setPrimaryKey(orderNumber);
```

6. Create foreign keys for the tables that need relationships. In this example, orders have a foreign key that points to the customer who made the order. In order to create a relationship between the two tables, you must first make a foreign key for the Orders table.

```
Key custFK = factory.createKey();
custFK.getColumns().add(custFKColumn);

orderTable.getForeignKeys().add(custFK);
```

The relationship takes two keys, the parent key and the child key. Because no specific name is given, the default concatenation of CUSTOMER_ORDER is the name used for this relationship.

```
metadata.addRelationship(custTable.getPrimaryKey(), custFK);
```

The default relationship includes all customers who have orders. To get all customers, even if they do not have orders, you need this line as well:

```
metadata.getRelationship("CUSTOMER_ORDER")
                            .setExclusive(false);
```

Now that the two tables are related to one another you can add a filter to the Customer table to find customers with specific characteristics.

7. Specify any filters needed. In this example, set filters to the Customer table to find all the customers in a particular state, with a certain last name, who have made orders.

```
Filter filter = factory.createFilter();
filter.setPredicate("CUSTOMER.CUSTSTATE = ? AND CUSTOMER.CUSTLASTNAME = ?");

FilterArgument arg1 = factory.createFilterArgument();
arg1.setName("CUSTSTATE");
arg1.setType(Column.STRING);
filter.getFilterArguments().add(arg1);

FilterArgument arg2 = factory.createFilterArgument();
arg2.setName("CUSTLASTNAME");
```

```
arg2.setType(Column.STRING);
filter.getFilterArguments().add(arg2);

custTable.setFilter(filter);
```

8. Add any order by objects needed. In this example, set the order by object to sort by the customer's first name.

```
Column firstName = ((TableImpl)custTable).getColumn("CUSTFIRSTNAME");
OrderBy orderBy = factory.createOrderBy();
orderBy.setColumn(firstName);
orderBy.setAscending(true);
metadata.getOrderBys().add(orderBy);
```

This completes the creation of the metadata for this JDBC DMS.

9. Create a connection to the database. This example does not show the creation of the connection to the database; it assumes that the SDO client calls the method connect() that does that. See the topic,
10. Instantiate and initialize the JDBC DMS object (dataGraph). The SDO client performs these actions. For this example:

```
ConnectionWrapperFactory factory = ConnectionWrapperFactory.soleInstance;
connectionWrapper = factory.createConnectionWrapper(connect());
JDBCMediatorFactory mFactory = JDBCMediatorFactory.soleInstance;
JDBCMediator mediator = mFactory.createMediator(metadata, connectionWrapper);
DataObject parameters = mediator.getParameterDataObject();
parameters.setString("CUSTSTATE", "NY");
parameters.setString('CUSTLASTNAME', 'Smith');
DataObject graph = mediator.getGraph(parameters);
```

Now that you have the dataGraph, you can manipulate the information. The example below contains basic manipulation of data in a DataGraph object.

Example: Manipulating data in a DataGraph object

Using the simple DataGraph that was created during the task Using the Java Database Connectivity data meditaor service for data access, some typical data manipulation follows.

First get the list of customers, then for each customer get every order, then print out the customer's first name and order date. (For this example, assume that you already know the last name is Smith).

```
List customersList = graph.getList("CUSTOMER");
Iterator i = customersList.iterator();
while (i.hasNext())
{
 DataObject customer = (DataObject)i.next();
List ordersList = customer.getList("CUSTOMER_ORDER");
Iterator j = ordersList.iterator();
while (j.hasNext())
{
DataObject order = (DataObject)j.next();
System.out.print( customer.get("CUSTFIRSTNAME") + " ");
System.out.println( order.get("ORDERDATE"));
 }
}
```

Now change every customer with the name Will to be Matt.

```
i = customersList.iterator();
while (i.hasNext())
{
 DataObject customer = (DataObject)i.next();
 if (customer.get("CUSTFIRSTNAME").equals("Will"))
 {
  customer.set("CUSTFIRSTNAME", "Matt");
 }
}
```

Delete the first Customer entry.

```
((DataObject) customersList.get(0)).delete();
```

Add a new DataObject to the graph

```
DataObject newCust = graph.createDataObject("CUSTOMER");
newCust.setInt("CUSTID", 12345);
newCust.set("CUSTFIRSTNAME", "Will");
newCust.set("CUSTLASTNAME", "Smith");
newCust.set("CUSTSTREETADDRESS", "123 Main St.");
newCust.set("CUSTCITY", "New York");
newCust.set("CUSTSTATE", "NY");
newCust.set("CUSTZIPCODE", "12345");
newCust.setInt("CUSTAREACODE", 555);
newCust.set("CUSTPHONENUMBER", "555-5555");

graph.getList("CUSTOMER").add(newCust);
```

Submit the changes.

```
mediator.applyChanges(graph);
```

11. Submit the changed information to the DMS for updating the database.

Using the EJB data mediator service for data access

The following steps use code samples to describe a simple instance of how to create the Enterprise JavaBeans (EJB) data mediator service (DMS) metadata.

Procedure

1. A mediator instance is created using one of the create methods on the mediator factory (com.ibm.websphere.sdo.mediator.ejb.MediatorFactory) as in the following example

```
import com.ibm.websphere.sdo.mediator.ejb.Mediator;
import com.ibm.websphere.sdo.mediator.ejb.MediatorFactory;
import com.ibm.webpshere.ejbquery.QueryException;
import commonj.sdo.DataObject;

try{
 String[] query =  { "select d.deptno,d.name from DeptBean as d" };
 Mediator m = MediatorFactory.getInstance().createMediator( query, null);
 DataObject root = m.getGraph();
} catch (QueryException e) { ... }
```

2. There are 3 different forms of the createMediator method. The arguments are explained as follows:

```
createMediator( query, parms)
createMediator( query, parms, schema )
createMediator( query, parms, schema,  typeMap,  pattern)
```

Table 11. createMediator method arguments. The arguments are explained in the following table:

Type	Argument	Description
String	query	array of EJB query statements
Object	parms	values for input parameters of the query statements
Eclass*	schema	the EClass of the root DataObject
Map*	typeMap	a java.util.Map that maps EJB Abstract Schema Names from the query statement into Eclass names
int*	pattern	the pattern used for containment
* used only when using caller provided schema		

Example

Using query arguments with EJB mediator.

The following examples show how you can fine-tune your Enterprise JavaBeans (EJB) mediator query arguments.

A simple example

This query returns a DataGraph containing multiple instances of DataObjects of type (Eclass name) *Emp*. The data object attributes are *empid* and *name* and their data types correspond to the container-managed persistence (CMP) field types.

```
select  e.empid, e.name
from Emp as e
where e.salary > 100
```

The returned DataGraph serialized in its XML format looks like this:

```
  <?xml version="1.0" encoding="ASCII"?>
<datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
<root>
<Emp empid="1003" name="Eric" />
<Emp empid="1004" name="Dave" />
</root>
</datagraph:DataGraphSchema>
```

Query parameters

This example shows how parameter markers can be used. Recall that the syntax for parameter markers in an EJB query is a question mark followed by a number (?n). When calling the getGraph ( ) method on the EJBMediator, you can optionally pass an array of values. ?n refers to the value of parm[n-1]. The array of values can also be passed on the factory call to create the EJBMediator. Parameters passed on the getGraph( ) override any parameters passed on the create call.

```
select  e.empid, e.name
from Emp as e
where e.salary > ?1
```

Returning expressions and methods

This example illustrates that the data object attributes can be the return values of query expressions. EJB query expressions include arithmetic, date-time, path expressions, and methods. Input arguments and return values from methods are restricted to the list of supported data types see the topic EJB mediator query syntax. A data object containing an updated attribute derived from an expression causes an exception to occur during the applyChanges process unless the user has provided a MediatorAdapter to handle the change.

```
select  e.empid as employeeId,
  e.bonus+e.salary as totalPay,
  e.dept.mgr.name as managerNam,
  e.computePension( ) as pension
from Emp as e
where e.salary > 100
```

Data object attribute names are derived from the CMP field names but can be overridden by using the *AS* keyword in the query. When specifying an expression, the *AS* keyword should always be used to give a name to the expression.

The * syntax

The notation *e.** is a short cut for specifying all the CMP fields (but not container-managed relationships) for an EJB. The following query means the same thing as *e.empid, e.name e.salary, e.bonus*.

```
select e.* from Emp as e
```

No primary key in select clause

This example shows a query that does not return the primary key field. However, unless the data object contains all the primary key fields for an EJB, updates to the DataGraph cannot be processed by the mediator. This is because the primary key is required to translate the changes into structured query language (SQL), or to convert DataObject references to EJB references. An exception when applyChanges tries to run.

```
select e.name, e.salary from Emp as e
```

Order by

DataObjects can be ordered.

```
select d.* from Dept d  order by d.name
 select e.* from in(d.emps) e order by e.empid desc
```

This results in the *Dept* objects being ordered by name and the *Emp* objects within each *Dept* being order by *empid* in descending order.

Navigating a multi-valued relationship

This compound query returns a DataGraph with DataObject classes *Dept* and *Emp*. The shape of the DataGraph reflects the path expressions used in the FROM clauses.

```
select d.deptno, d.name, d.budget from Dept d
   where d.deptno < 10
select e.empid, e.name, e.salary  from in(d.emps) e
   where e.salary > 10
```

In this case *Dept* is the root node in the DataGraph and there is a multivalued reference from *Dept* to *Emp* as shown:

```
<?xml version="1.0" encoding="ASCII" ?>
<datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
<root>
<Dept deptno="1" name="WAS_Sales" budget="500.0"
   emps="//@root/@Emp.1 //@root/@Emp.0" />
<Dept deptno="2" name="WBI_Sales" budget="450.0"
   emps="//@root/@Emp.3 //@root/@Emp.2" />
<Emp empid="1001" name="Rob" salary="100.0" EmpDept="//@root/@Dept.0" />
<Emp empid="1002" name="Jason" salary="100.0" EmpDept="//@root/@Dept.0" />
<Emp empid="1003" name="Eric" salary="200.0" EmpDept="//@root/@Dept.1" />
<Emp empid="1004" name="Dave" salary="500.0" EmpDept="//@root/@Dept.1" />
</root>
</datagraph:DataGraphSchema>
```

More on query parameters

Search conditions can be specified on any query. Input arguments are global to the query and can be referenced by number anywhere in the compound query. In the example above, the query arguments passed on the create or getGraph call should be in order { deptno value, salary value, deptno value }.

```
select d.* from Dept as d
   where d.deptno between ?1 and ?3
select e.* from in(d.emps) e
   where e.salary < ?2
```

Navigating a path with multiple relationships

The following query navigates the path composed of EJB relationships Dept.projs and Project.tasks and returns DataObjects for Dept, Emp and Project containing selected CMP fields.

```
select d.deptno, d.name from Dept as d
select p.projid  from in(d.projects) p
select t.taskid, t.cost from  in (p.tasks) t
```

The resulting data graph in XML format is shown here.

```
<?xml version="1.0" encoding="ASCII" ?>
<datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
<root>
<Dept deptno="1" name="WAS_Sales" projects="//@root/@Project.0" />
<Dept deptno="2" name="WBI_Sales" projects="//@root/@Project.1" />
<Project projid="1" ProjectDept="//@root/@Dept.0"
      tasks="//@root/@Task.0   //@root/@Task.2 //@root/@Task.1" />
<Project projid="2" ProjectDept="//@root/@Dept.1"
      tasks="//@root/@Task.3" />
<Task taskid="1" cost="50.0" TaskProject="//@root/@Project.0" />
<Task taskid="2" cost="60.0" TaskProject="//@root/@Project.0" />
<Task taskid="3" cost="900.0" TaskProject="//@root/@Project.0" />
<Task taskid="7" cost="20.0" TaskProject="//@root/@Project.1" />
</root>
</datagraph:DataGraphSchema>
```

Navigating multiple paths

Here is a mediator query returning a DataGraph with DataObjects for Dept with related employees and a second path that retrieves related projects and tasks.

```
select d.deptno, d.name from Dept d
select e.empid, e.name  from in(d.emps) e
select p.projid from in(d.projects) p
select t.taskid, t.cost from in(p.tasks) where t.cost > 10
```

The returned DataGraph looks like this:

```
<?xml version="1.0" encoding="ASCII" ?>
    <datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
    <root>
        <Dept deptno="1" name="WAS_Sales" projects="//@root/@Project.0"
 emps="//@root/@Emp.1 //@root/@Emp.0" />
        <Dept deptno="2" name="WBI_Sales" projects="//@root/@Project.1"
     emps="//@root/@Emp.3 //@root/@Emp.2" />
        <Project projid="1" ProjectDept = "//@root/@Dept.0"
     tasks="//@root/@Task.0 //@root/@Task.2 //@root/@Task.1" />
        <Project projid="2" ProjectDept="//@root/@Dept.1" tasks="//@root/@Task.3" />
      <Task taskid="1" cost="50.0" TaskProject="//@root/@Project.0" />
        <Task taskid="2" cost="60.0" TaskProject="//@root/@Project.0" />
        <Task taskid="3" cost="900.0" TaskProject="//@root/@Project.0" />
        <Task taskid="7" cost="20.0" TaskProject="//@root/@Project.1" />
        <Emp empid="1001" name="Rob" EmpDept="//@root/@Dept.0" />
        <Emp empid="1002" name="Jason" EmpDept="//@root/@Dept.0" />
         <Emp empid="1003" name="Eric" EmpDept="//@root/@Dept.1" />
         <Emp empid="1004" name="Dave" EmpDept="//@root/@Dept.1" />
  </root>
  </datagraph:DataGraphSchema>
```

Navigating a single valued relationship

The important thing to point out here is that even though Emp is the root data object in the graph, multiple Emp data objects will be related to the same Dept data object. So unlike the previous examples, the data graph does not have a tree shape when you look at the data object instances – there are multiple root Emp objects related to the same Dept object. But then after all it is a data graph, not a data tree. Note that mediator queries allow single valued path expressions in the FROM clause. This is a change from the standard EJB query syntax.

```
select e.empid, e.name  from Emp e
select d.deptno, d.name  from in(e.dept) d
```

And the DataGraph in XML format looks like:

```
<?xml version="1.0" encoding="ASCII" ?>
   <datagraph:DataGraphSchema xmlns:datagraph="datagraph.ecore">
   <root>
   <Emp empid="1001" name="Rob" dept="//@root/@Dept.0" />
```

```
   <Emp empid="1002" name="Jason" dept="//@root/@Dept.0" />
   <Emp empid="1003" name="Eric" dept="//@root/@Dept.1" />
   <Emp empid="1004" name="Dave" dept="//@root/@Dept.1" />
   <Dept deptno="1" name="WAS_Sales"
DeptEmp="//@root/@Emp.1 //@root/@Emp.0" />
   <Dept deptno="2" name="WBI_Sales"
 DeptEmp="//@root/@Emp.3 //@root/@Emp.2" />
 </root>
 </datagraph:DataGraphSchema>
```

Path expressions in the SELECT clause

This query is similar to the preceding one (both queries return employee data along with department number and name) but note the data graph contains only one data object type in this query (vs. two in the previous query). The fields deptno and name field are read only because they are result of a path expression in the SELECT clause and are not CMP fields of the Emp EJB.

```
select  e.empid as EmplId , e.name as EmpName ,
   e.dept.deptno as DeptNo , e.dept.name as DeptName
from Emp as e
```

Navigating a many: many relationship

The Emp to Task relationship is deemed a *many:many* relationship. The following query retrieves employees, tasks, and projects. There is only a single occurrence of any particular task DataObject in the DataGraph, even though it can be related to many employees.

```
select e.empid, e.name from Emp as e
select t.taskid, t.description from in(e.tasks) as t
select p.projid, p.cost from in(t.proj) as p
```

Multiple links between data objects

The EJB mediator enables you to retrieve data based on relationships and use the XREL command to construct one or more additional relationships based on data already retrieved. The mediator also enables retrieval of data based on ASNname and then construction of one or more relationships based on the data retrieved using the XREL command. The following query retrieves departments, employees that work in the departments, and the employees that manage the departments.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from in (d.emps) as e
select m.empid, m.name from in(d.manager) as m
```

The second and third *select* clauses both return instances of Emp DataObject. It is possible that the same Emp instance is retrieved through the d.emps relationship and the d.manager relationship. The EJB mediator creates one Emp instance, but creates both relationships.

The following query is processed as follows. Dept DataObjects are created from the data in the first query. Emp DataObjects are created from the data in the second query. Relationships in the graph are then constructed for any relationship used in either the FROM clause or an XREL keyword. During relationship construction, no additional data is retrieved. In this example, an employee who works in a department named *Dev* appears in the DataGraph. If this employee manages a department called *Sales*, the *manages* reference is empty. The Dev department was retrieved in the first query, not the Sales department.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from in (d.emps) as e
xrel d.manager
```

The emps and manager relationship are constructed based on the DataObject instances created from the queries. An employee whose name is ‘Dev' but works in department ‘Sales' will have a null dept relationship in the graph.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from Emp e where e.name like 'Dev%'
xrel d.emps, d.manager
```

The next example shows the retrieval of data objects for all the employees, projects, and tasks for a given department, and the linkage of employees with tasks.

```
select d.deptno from Dept d where d.deptno = 42
select e.empid from in(d.emps) e
select p.projid from in(d.projs) p
select t.* from in(p.tasks) t
  xrel e.tasks
```

If a task is assigned to an employee in department 42 then that link appears in the data graph. If the task is assigned to an employee not in department 42, then that link does not appear in the data graph because the data object was filtered out by the query. An XREL keyword can be followed by one or more EJB relationships. Bidirectional relationships can refer to either role name. Both source and target of the relationship must be retrieved by one or more queries.

Retrieving unrelated objects

The following query retrieves Dept and Task.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select t.taskid, t.startDate from Task t where t.startDate > '2005'
```

The following query retrieves Dept and Emps. Even though there are relationships between Dept and Emp (namely mgr and emps), neither relationship is used in FROM or XREL and so the resulting graph does not contain the relationship values.

```
select d.deptno, d.name from Dept d where d.name like '%Dev%'
select e.empid, e.name from Emp e where e.dept.name like '%Dev%'
```

Retrieving null or empty relationships

This query returns departments that have no employees and employees with no department. Presumably the application wants to assign the employees to one of the departments. The purpose of xrel is to define the e.dept relationship (and the inverse role d.emps) into the graph schema.

```
select d.deptno, d.name from Dept d where d.emps is empty
select e.empid, e.name from Emp e where e.dept is null
xrel e.dept
```

Collection Input parameter

A collection of enterprise beans can be passed as an input argument to the ejb mediator and referenced in the FROM clause. Using a collection parameter satisfies the requirement to construct a data graph from a user collection of already activated enterprise beans.

```
select d.deptno, d.name from ((Dept) ?1) as d
select e.empid, e.name from in(d.emps) as e where e.salary > 10
```

The above query iterates through the collection of Dept beans and related Emp beans applying the query predicates and constructing the data graph. Values will be obtained from current values of the beans. An example of a program using an ejb collection parameter.

```
// this method runs in an EJB context and within a transaction scope
public DataGraph myServiceMethod() {
      InitialContext ic = new InitialContext();
      DeptLocalHome deptHome = ic.lookup("java:comp/env/ejb/Dept");
      Integer deptKey = new Integer(10);
```

```
      DeptEJB dept = deptHome.findByPrimaryKey( deptKey));
      Iterator i = dept.getEmps().iterator();
      while (i.hasNext()) {
       EmpEJB e = (EmpEJB)i.next();
       e.setSalary( e.getSalary() * 1.10); // give everyone a 10% raise
      }

      // create the query collection parameter
      Collection c = new LinkedList();
      c.add(dept);
      Object[] parms = new Object[] { c};  // put ejb collection in parm array.

      // collection containing the dept EJB is passed to EJB Mediator

      String[] query = new String[]
          { "select d.deptno, d.name from ((Dept)?1 ) as d",
             "select e.empid, e.name, e.salary " +
                 " from in (d.employees) as e",
            "select p.projno, p.name from in (d.projects) as p" };

      Mediator m = EJBMediatorFactory.getInstance().createMediator(
 query, parms);
      DataGraph dg = m.getGraph();
      return dg;
      // the DataGraph contains the updated and as yet uncommitted
//  salary information.  Dept and Emp data
      // is fetched through EJB instances active in the EJBContainer.
      //  Project data is retrieved from database using
      //   container managed relationships.
   }
```

Using MediatorAdapter

In this example, the adapter processes CREATE events for an EMP data object. The name and salary attributes are extracted from the data object and passed to the create method on the EmpLocalHome.

The create method returns an instance of Emp EJB and the primary key value is copied back to the DataObject. The caller can then obtain the generated key value. After processing, the adapter returns a value of **true**. All other changes are ignored by the adapter and processed by the EJB Mediator.

```
package com.example;
import com.ibm.websphere.sdo.mediator.ejb.*;
import javax.naming.InitialContext;
import commonj.sdo.ChangeSummary;
import commonj.sdo.DataObject;
import commonj.sdo.DataGraph;
import commonj.sdo.ChangeSummary;

// example of Adapter class calling a EJB create method.

public class SalaryAdapter implements MediatorAdapter{

 ChangeSummary log = null;
      EmpLocalHome empHome = null;

 public boolean applyChange(DataObject object,  int phase){

  if (object.getType().getName().equals("Emp")
       && phase == MediatorAdapter.CREATE){
   try{
    String name = object.getString("name");
    double salary = object.getDouble("salary");
    EmpLocal emp = empHome.create(name, salary);
    object.set("empid", emp.getPrimaryKey() ); // set primary key in SDO
    return true;
```

```
  } catch(Exception e){ // error handling code goes here
                  }
 }
 return true;
}

public void init (ChangeSummary log){
        try {
 this.log = log;
 InitialContext ic = new InitialContext();
 empHome = (EmpLocalHome)ic.lookup( "java:comp/env/ejb/Emp");
        } catch (Exception e) { // error handling code goes here
        }
}

public void end(){}
}
```

Developing a custom DataStoreHelper class

Apply the WebSphere extension, GenericDataStoreHelper class, to create your own data store helper for data sources that the application server does not support. With this helper class, your JDBC configuration can use database-specific functions during transactions.

Before you begin

If you are using a configuration with a data source that is not supported by the application server, you might want to create a custom data store helper. This helper will allow you to leverage the database to perform functions during a transaction that would not otherwise be available. You will need to create a user-defined DataStoreHelper class, and there is information for creating a new exception handler to catch any exceptions that might be created with the use of your custom data handler.

About this task

Note: The CUSTOM_HELPER constant field in the com.ibm.websphere.rsadapter.DataStoreHelper class API is deprecated. If you create your own DataStoreHelper implementation class, do not invoke setHelperType(DataStoreHelper.CUSTOM_HELPER). Instead, let the HelperType value be set by the implementation class from which it inherits.

Procedure

1. Create a class that extends the existing data store helpers. Use the following code as an example; this type of data source is based on a user-defined JDBC provider:

```
package com.ibm.websphere.examples.adapter;

import java.sql.SQLException;
import javax.resource.ResourceException;

import com.ibm.websphere.appprofile.accessintent.AccessIntent;
import com.ibm.websphere.ce.cm.*;
import com.ibm.websphere.rsadapter.WSInteractionSpec;

/**
* Example DataStoreHelper class, demonstrating how to create a user-defined DataStoreHelper.
* The implementation for each method is provided only as an example.  More detail is probably
* required for any custom DataStoreHelper that is created for use by a real application.
* In this example, we will override the doStatementCleanup(),getIsolationLevel(), and set userDefined
* exception map.
*/
public class ExampleDataStoreHelper extends com.ibm.websphere.rsadapter.GenericDataStoreHelper
{

    public ExampleDataStoreHelper(java.util.Properties props)
    {
        super(props);

        // Update the DataStoreHelperMetaData values for this helper.
        getMetaData().setGetTypeMapSupport(false);

        // Update the exception mappings for this helper.
        java.util.Map xMap = new java.util.HashMap();
```

```
        // Add an Error Code mapping to StaleConnectionException.
        xMap.put(new Integer(2310),  StaleConnectionException.class);
        // Add an Error Code mapping to DuplicateKeyException.
        xMap.put(new Integer(1062),  DuplicateKeyException.class);
        // Add a SQL State mapping to the user-defined ColumnNotFoundException
        xMap.put("S0022",            ColumnNotFoundException.class);
        // Undo an inherited StaleConnection SQL State mapping.
        xMap.put("S1000",            Void.class);

        setUserDefinedMap(xMap);

        // If you are extending a helper class, it is
        // normally not necessary to issue 'getMetaData().setHelperType(...)'
        // because your custom helper will inherit the helper type from its
        //  parent class.


         }

    public void doStatementCleanup(java.sql.PreparedStatement stmt) throws SQLException
    {
        // Clean up the statement so it may be cached and reused.

        stmt.setCursorName("");
        stmt.setEscapeProcessing(true);
        stmt.setFetchDirection(java.sql.ResultSet.FETCH_FORWARD);
        stmt.setMaxFieldSize(0);
        stmt.setMaxRows(0);
        stmt.setQueryTimeout(0);
    }



    public int getIsolationLevel(AccessIntent intent) throws ResourceException
    {
        // Determine an isolation level based on the AccessIntent.

        // set WebSphere default isolation level to TRANSACTION_SERIALIZABLE.
        if (intent == null) return java.sql.Connection.TRANSACTION_SERIALIZABLE;

        return intent.getConcurrencyControl() == AccessIntent.CONCURRENCY_CONTROL_OPTIMISTIC ?
               java.sql.Connection.TRANSACTION_READ_COMMITTED :
               java.sql.Connection.TRANSACTION_REPEATABLE_READ;
    }
        public int getLockType(AccessIntent intent) {
           if ( intent.getConcurrencyControl() == AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC) {
              if ( intent.getAccessType() == AccessIntent.ACCESS_TYPE_READ ) {
                  return WSInteractionSpec.LOCKTYPE_SELECT;
              }
              else {
                  return WSInteractionSpec.LOCKTYPE_SELECT_FOR_UPDATE;
               }
           }
           return WSInteractionSpec.LOCKTYPE_SELECT;
        }
}
```

2. Optional: Create your own exception handler class. Use the following code as a guide:

```
package com.ibm.websphere.examples.adapter;

import java.sql.SQLException;
import com.ibm.websphere.ce.cm.PortableSQLException;

/**
* Example PortableSQLException subclass, which demonstrates how to create a user-defined
* exception for exception mapping.
*/
public class ColumnNotFoundException extends PortableSQLException
{
    public ColumnNotFoundException(SQLException sqlX)
    {
        super(sqlX);
    }
}
```

3. Compile the newly created DataStoreHelper class or classes. You will need the following JAR files in your classpath to compile them:
   - *app_server_root*/dev/JavaEE/j2ee.jar
   - *app_server_root*/dev/was_public.jar

     Note: was_public.jar contains classes from *app_server_root*/lib/rsahelpers.jar that are needed to compile a custom DataStoreHelper class.
   - *app_server_root*/plugins/com.ibm.ws.runtime.jar

- If you are using a development environment, such as Eclipse, you need to set the above JAR files in your classpath to be able to compile. Then, create a JAR file of the project after you have finished editing your files (see the help documentation for your development environment for specific instructions).
- If you do not have development environment, and you are using the javac compiler:
  a. Create your .java file that extends the GenericDataStoreHelper or any other data store helper, as shown in Step 1.
  b. Change to your home directory after you are done editing your file or files in the command line utility.
  c. Set the classpath using this command:
  d. Compile your class or classes. For example, on Windows operating systems enter the following command (this will compile all the .java files in the directory that you specify):

     ```
     C:\javac your_directory\*.java
     ```

  e. From the `Java` directory, create a JAR file of all the compiled class files in your directory. For example, enter the following command on Windows operating systems (change *myFile* to the name you want for your JAR file):

     ```
     C:\Java> jar cf myFile.jar *.class
     ```

     For more information on using the javac compiler go to the Oracle website for the Java compiler.

4. Place your compiled JAR files in a directory, and update the class path for the JDBC provider to include that location. For example, if your JAR file is `c:\myFile.jar`, then make sure to modify the JDBC class path to include c:\myFile.jar.
   a. Click **Resources** > **JDBC** > **JDBC Providers** > ***JDBC_provider***.
   b. In the **Class path** field, add the location of the JAR files that you compiled. For example, press ENTER in the field and add a new line:

      ```
      c:\myFile.jar
      ```

5. Configure the application server to use your new custom DataStoreHelper class.
   a. From the administrative console select **Resources** > **JDBC** > **Data Sources**.
   b. Select the data source that you want to configure with your custom DataStoreHelper class.
   c. In the section labeled **Data store helper class name**, select **Specify a user-defined data store helper**.
   d. Enter the class name for the data store helper that you created.
   e. Apply your changes and select **OK**.

Example: Setting client information with the setClientInformation(Properties) API

You can set WebSphere Application Server client information on connections to pass that information to your database with this API.

The following example code calls setClientInformation(Properties) on the com.ibm.websphere.rsadapter.WSConnection object.

```
import com.ibm.websphere.rsadapter.WSConnection;
.....
try {
   InitialContext ctx = new InitialContext();
   //Perform a naming service lookup to get the DataSource object.
   DataSource ds = (javax.sql.DataSource)ctx.lookup("java:comp/jdbc/myDS");
  }catch (Exception e) {System.out.println("got an exception during lookup:"  + e);}

  WSConnection conn = (WSConnection) ds.getConnection();
  Properties props = new properties();
  props.setProperty(WSConnection.CLIENT_ID, "user123");
  props.setProperty(WSConnection.CLIENT_LOCATION, "127.0.0.1");
```

```
props.setProperty(WSConnection.CLIENT_ACCOUNTING_INFO, "accounting");
props.setProperty(WSConnection.CLIENT_APPLICATION_NAME, "appname");
props.setProperty(WSConnection.CLIENT_OTHER_INFO, "cool stuff");
conn.setClientInformation(props);
conn.close()
```

Parameters

props contains the client information to be passed. Possible values are:

- WSConnection.CLIENT_ACCOUNTING_INFO
- WSConnection.CLIENT_LOCATION
- WSConnection.CLIENT_ID
- WSConnection.CLIENT_APPLICATION_NAME
- WSConnection.CLIENT_OTHER_INFO
- WSConnection.OTHER_CLIENT_TYPE

Refer to the WSConnection documentation for more details on which client information is passed to the backend database. To reset the client information, call the method with a null parameter.

Exceptions

This API creates an SQL exception if the database issues an exception when setting the data.

Passing client info to a db cdat_clientinfo

Changing the error detection model to use the Exception Checking Model

The error detection model has been expanded and the data source has a configuration option that you can use to select the exception mapping model or the exception checking model for error detection. This configuration option allows the Error Detection Model to comply with Java Database Connectivity (JDBC) 4.0.

About this task

By default, the exception mapping Error Detection Model configuration is selected. The exception mapping Error Detection Model replaces some exceptions raised by the JDBC driver. Exception checking does not do this. If you want to use this configuration, no changes are needed. If you want to use the exception checking model, you need to configure the error detection model in the application server. If you previously changed the **Error Detection Model**, you can also use these steps to change the configuration back to using to the exception mapping model.

Procedure

1. Open the administrative console.
2. Go to the **WebSphere Application Server Data Source properties** panel for the data source.
   a. Select **Resources** > **JDBC** > **Data Sources** > ***data_source***
   b. Select **WebSphere Application Server Data Source properties**.
3. In the **Error Detection Model** section, click **Use the WebSphere Application Server Exception Checking Model**.

Exceptions pertaining to data access

All enterprise bean container-managed persistence (CMP) beans under the Enterprise JavaBeans (EJB) 2.x specification receive a standard EJB exception when an operation fails. Java Database Connectivity

(JDBC) applications receive a standard SQL exception if any JDBC operation fails. The product provides special exceptions for its relational resource adapter (RRA), to indicate that the connection currently held is no longer valid.

- The connection wait timeout exception indicates that the application has waited for the number of seconds specified by the connection timeout setting and has not received a connection. This situation can occur when the pool is at maximum size and all of the connections are in use by other applications for the duration of the wait. In addition, there are no connections currently in use that the application can share because either the connection properties do not match, or the connection is in a different transaction.

  For a Version 4.0 data source, the ConnectionWaitTimeout object creates an exception that is instantiated from the com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException class.

  For Java 2 Connector (J2C) connection factories, the ConnectionWaitTimeout object generates a resource exception of the com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException class.

  When the error detection model is configured to exception mapping, later versions of data sources issue an SQL exception of the com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException subclass. When the error detection model is configured to exception checking, later versions of data sources issue an SQL exception of the java.sql.SQLTransientConnectionException class with a chained exception of the com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException class.
- When the error detection model is configured to exception mapping, the stale connection exception indicates that the connection is no longer valid. When the error detection model is configured to exception checking, the JDBC driver raises a JDBC 4.0 exception, such as java.sql.SQLRecoverableException or java.sql.SQLNonTransientConnectionException, or the JDBC driver specifies an appropriate SQLState to indicate that the connection is no longer valid. Read the topic Stale connections for more information about this type of exception.

Note: The userDefinedErrorMap custom property overlays existing entries in the error map by starting the DataStoreHelper.setUserDefinedMap method. You can use the custom property to add, change, or remove entries from the error map.

- Entries are delimited by a ; (semicolon).
- Each entry consists of a key and value, where the key is an error code (numeric value) or SQLState, which is text enclosed in quotation marks.
- Keys and values are separated by the = (equals sign).

For example, to remove the mapping of SQLState S1000, add a mapping of error code 1062 to duplicate key, and add a mapping of SQLState 08004 to stale connection, you can specify the following value for userDefinedErrorMap:

```
"S1000"=;1062=com.ibm.websphere.ce.cm.DuplicateKeyException;"08004"=
com.ibm.websphere.ce.cm.StaleConnectionException
```

userDefinedErrorMap can be located in the administrative console by selecting the data source and configuring the custom properties.

Examples Table of Contents

- Stale Connections
- Example: Handling data access exception - stale connection
- Stale Connection on Linux systems
- Example: Handling servlet JDBC connection exceptions
- Example: Handling connection exceptions for session beans in container-managed database transactions
- Example: Handling connection exceptions for session beans in bean-managed database transactions
- Example: Handling connection exceptions for BMP beans in container-managed database transactions
- Example: Handling data access exception - ConnectionWaitTimeoutException (for the JDBC API)

- Example: Handling data access exception - ConnectionWaitTimeoutException (for Java EE Connector Architecture)
- Example: Handling data access exception - error mapping in DataStoreHelper
- Database deadlock and foreign key conflicts

Stale Connections

The product provides a special subclass of the java.sql.SQLException class for using connection pooling to access a relational database. This com.ibm.websphere.ce.cm.StaleConnectionException subclass exists in both a WebSphere 4.0 data source and in the most recent version data source that use the relational resource adapter. This class serves to indicate that the connection currently held is no longer valid.

This situation can occur for many reasons, including the following:
- The application tries to get a connection and fails, as when the database is not started.
- A connection is no longer usable because of a database failure. When an application tries to use a previously obtained connection, the connection is no longer valid. In this case, all connections currently in use by the application can get this error when they try to use the connection.
- The connection is orphaned (because the application had not used it in at most two times the value of the unused timeout setting) and the application tries to use the orphaned connection. This case applies only to Version 4.0 data sources.
- The application tries to use a JDBC resource, such as a statement, obtained on a stale connection.
- A connection is closed by the Version 4.0 data source auto connection cleanup feature and is no longer usable. Auto connection cleanup is the standard mode in which connection management operates. This mode indicates that at the end of a transaction, the transaction manager closes all connections enlisted in that transaction. This enables the transaction manager to ensure that connections are not held for excessive periods of time and that the pool does not reach its maximum number of connections prematurely.

  A negative ramification does ensue, however, when the transaction manager closes the connections and returns the connection to the free pool after a transaction ends. An application cannot obtain a connection in one transaction and try to use it in another transaction. If the application tries this connection, a stale connection exception occurs because the connection is already closed.

If you are trying to use an orphaned connection or a connection that is made unavailable by auto connection cleanup, a stale connection exception indicates that the application has attempted to use a connection that is already returned to the connection pool. It does not indicate an actual problem with the connection. However, other cases of a stale connection exception indicate that the connection to the database has gone bad, or stale. Once a connection has gone stale, you cannot recover it, and you must completely close the connection rather than returning it to the pool.

Detecting stale connections

When a connection to the database becomes stale, operations on that connection result in an SQL exception from the JDBC driver. Because an SQL exception is a rather generic exception, it contains state and error code values that you can use to determine the meaning of the exception. However, the meanings of these states and error codes vary depending on the database vendor. The connection pooling run time maintains a mapping of which SQL state and error codes indicate a stale connection exception for each database vendor supported. When the connection pooling run time catches an SQL exception, it checks to see if this SQL exception is considered a stale connection exception for the database server in use.

Recovering from stale connections

An application can catch a stale connection exception, depending on the type of error detection model that is configured on the data source:

- When the error detection model is configured to exception mapping, the application server replaces the exception that is raised by the JDBC driver with StaleConnectionException. In this case, the application might trap for a stale connection exception.
- When the error detection model is configured to exception checking, the application server still consults the error map in order to manage the connection pool, but it does not replace the exception. In this case, the application should not trap for a stale connection exception.

Because of the differences between error detection models, the application server provides an API that applications can use with either case to identify stale connections. The API is com.ibm.websphere.rsadapter.WSCallHelper.getDataStoreHelper(datasource).isConnectionError(sqlexception).

Applications are not required to explicitly identify a stale connection exception. Applications are already required to catch the java.sql.SQLException, and the stale connection exception or the exception that is raised by the JDBC driver, always inherits data from the java.sql.SQLException. The stale connection exception, which can result from any method that is declared to raise SQLException, is caught automatically in the general catch-block. However, explicitly identifying a stale connection exception makes it possible for an application to recover from bad connections. When application code identifies a stale connection exception, it should take explicit steps to recover, such as retrying the operation under a new transaction and new connection.

Example: Handling data access exception - stale connection

These code samples demonstrate how to programmatically address stale connection exceptions for different types of data access clients in different transaction scenarios.

When an application receives a stale connection exception on a database operation, it indicates that the connection currently held is no longer valid. Although it is possible to get an exception for a stale connection on any database operation, the most common time to see a stale connection exception issued is after the first time the connection is retrieved. Because connections are pooled, a database failure is not detected until the operation immediately following its retrieval from the pool, which is the first time communication to the database is attempted. It is only when a failure is detected that the connection is marked stale. The stale connection exception occurs less often if each method that accesses the database gets a new connection from the pool.

Many stale connection exceptions are caused by intermittent problems with the network of the database server. Obtaining a new connection and retrying the operation can result in successful completion without exceptions. In some cases it is advantageous to add a small wait time between the retries to give the database server more time to recover. However, applications should not retry operations indefinitely, in case the database is down for an extended time.

Note: If you are developing applications for the Application Server with an integrated development environment (IDE) like Eclipse, you might must import the *app_server_root*/`plugins/com.ibm.ws.runtime.jar` file into the development environment to take advantage of code that is provided.

Before the application can obtain a new connection for a retry of the operation, roll back the transaction in which the original connection was involved and begin a new transaction. You can break down details on this action into the following two categories:

Objects operating in a bean-managed global transaction context begun in the same method as the database access

A servlet or session bean with bean-managed transactions (BMT) can start a global transaction explicitly by calling `begin()` on a `javax.transaction.UserTransaction` object, which you can retrieve from naming or from the bean `EJBContext` object. To commit a bean-managed transaction, the application calls `commit()` on the `UserTransaction` object. To roll back the transaction, the application calls `rollback()`. Entity beans and non-BMT session beans cannot explicitly begin global transactions.

If an object that explicitly started a bean-managed transaction receives a stale connection exception on a database operation, close the connection and roll back the transaction. At this point, the application developer can decide to begin a new transaction, get a new connection, and retry the operation.

The following code fragment shows an example of handling stale connection exceptions in this scenario:

```
//get a userTransaction
javax.transaction.UserTransaction tran = getSessionContext().getUserTransaction();
//retry indicates whether to retry or not
//numOfRetries states how many retries have
// been attempted
boolean retry = false;
int numOfRetries = 0;
java.sql.Connection conn = null;
java.sql.Statement stmt = null;
do {
  try {
    //begin a transaction
    tran.begin();
    //Assumes that a datasource has already been obtained
    //from JNDI
    conn = ds.getConnection();
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    stmt.execute("INSERT INTO EMPLOYEES VALUES
              (0101, 'Bill', 'R', 'Smith')");
    tran.commit();
    retry = false;
  } catch(java.sql.SQLException sqlX)
  {
    // If the error indicates the connection is stale, then
    // rollback and retry the action
    if (com.ibm.websphere.rsadapter.WSCallHelper
        .getDataStoreHelper(ds)
        .isConnectionError(sqlX))
    {
      try {
        tran.rollback();
      } catch (java.lang.Exception e) {
        //deal with exception
        //in most cases, this can be ignored
      }
      if (numOfRetries < 2) {
        retry = true;
        numOfRetries++;
      } else {
        retry = false;
      }
    }
    else
    {
      //deal with other database exception
      retry = false
    }
  } finally {
    //always cleanup JDBC resources
    try {
      if(stmt != null) stmt.close();
    } catch (java.sql.SQLException sqle) {
      //usually can ignore
    }
    try {
      if(conn != null) conn.close();
    } catch (java.sql.SQLException sqle) {
      //usually can ignore
    }
  }
} while (retry) ;
```

Objects operating in a global transaction context and transaction not begun in the same method as the database access.

When the object which receives the stale connection exception does not have direct control over the transaction, such as in a container-managed transaction case, the object must mark the

transaction for rollback, and then indicate to its caller to retry the transaction. In most cases, you can do this by creating an application exception that indicates to retry that operation. However this action is not always allowed, and often a method is defined only to create a particular exception. This is the case with the ejbLoad() and ejbStore() methods on an enterprise bean. The next two examples explain each of these scenarios.

Example 1: Database access method creates an application exception

When the method that accesses the database is free to create whatever exception is required, the best practice is to catch the stale connection exception and create some application exception that you can interpret to retry the method. The following example shows an EJB client calling a method on an entity bean with transaction demarcation TX_REQUIRED, which means that the container begins a global transaction when *insertValue()* is called:

```
public class MyEJBClient
{
  //... other methods here ...

  public void myEJBClientMethod()
  {
    MyEJB myEJB = myEJBHome.findByPrimaryKey("myEJB");
    boolean retry = false;
    do
    {
      try
      {
        retry = false;
        myEJB.insertValue();
      }
      catch(RetryableConnectionException retryable)
      {
        retry = true;
      }
      catch(Exception e) { /* handle some other problem */ }
    }
    while (retry);
  }
}  //end MyEJBClient

public class MyEJB implements javax.ejb.EntityBean
{
  //... other methods here ...
  public void insertValue() throws RetryableConnectionException,
         java.rmi.EJBException
  {
    try
    {
      conn = ds.getConnection();
      stmt = conn.createStatement();
      stmt.execute("INSERT INTO my_table VALUES (1)");
    }
    catch(java.sql.SQLException sqlX)
    {
      // Find out if the error indicates the connection is stale
      if (com.ibm.websphere.rsadapter.WSCallHelper
          .getDataStoreHelper(ds)
          .isConnectionError(sqlX))
      {
        getSessionContext().setRollbackOnly();
        throw new RetryableConnectionException();
      }
      else
      {
        //handle other database problem
      }
    }
    finally
    {
      //always cleanup JDBC resources
      try
      {
        if(stmt != null) stmt.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
      try
      {
        if(conn != null) conn.close();
```

```
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
    }
  }
}  //end MyEJB
```

`MyEJBClient` first gets a `MyEJB` bean from the home interface, assumed to have been previously retrieved from the Java Naming and Directory Interface (JNDI). It then calls `insertValue()` on the bean. The method on the bean gets a connection and tries to insert a value into a table. If one of the methods fails with a stale connection exception, it marks the transaction for `rollbackOnly` (which forces the caller to roll back this transaction) and creates a new `retryable connection` exception, cleaning up the resources before the exception is thrown. The retryable connection exception is simply an application-defined exception that tells the caller to retry the method. The caller monitors the retryable connection exception and, if it is caught, retries the method. In this example, because the container is beginning and ending the transaction; no transaction management is needed in the client or the server. Of course, the client could start a bean-managed transaction and the behavior would still be the same, provided that the client also committed or rolled back the transaction.

Example 2: Database access method creates an onlyRemote exception or an EJB exception

Not all methods are allowed to throw exceptions defined by the application. If you use bean-managed persistence (BMP), use the `ejbLoad()` and `ejbStore()` methods to store the bean state. The only exceptions issued from these methods are the java.rmi.Remote exception or the javax.ejb.EJB exception, so you cannot use something similar to the previous example.

If you use container-managed persistence (CMP), the container manages the bean persistence, and it is the container that sees the stale connection exception. If a stale connection is detected, by the time the exception is returned to the client it is simply a remote exception, and so a simple catch-block does not suffice. There is a way to determine if the root cause of a remote exception is a stale connection exception. When a remote exception is created to wrap another exception, the original exception is usually retained. All remote exception instances have a detail property, which is of type `java.lang.Throwable`. With this detail, you can trace back to the original exception and, if it is a stale connection exception, retry the transaction. In reality, when one of these remote exceptions flows from one Java Virtual Machine API to the next, the detail is lost, so it is better to start a transaction in the same server as the database access occurs. For this reason, the following example shows an entity bean accessed by a session bean with bean-managed transaction demarcation.

```
public class MySessionBean extends javax.ejb.SessionBean
{
  ... other methods here ...
  public void mySessionBMTMethod() throws
    java.rmi.EJBException
  {
    javax.transaction.UserTransaction tran =
    getSessionContext().getUserTransaction();
    boolean retry = false;
    do
    {
      try
      {
        retry = false;
        tran.begin();
        // causes ejbLoad() to be invoked
        myBMPBean.myMethod();
        // causes ejbStore() to be invoked
        tran.commit();
      }
      catch(java.rmi.EJBException re)
      {
        try
        {
```

```
          tran.rollback();
        }
        catch(Exception e)
        {
          //can ignore
        }
        if (causedByStaleConnection(re))
          retry = true;
        else
          throw re;
      }
      catch(Exception e)
      {
        // handle some other problem
      }
      finally
      {
        //always cleanup JDBC resources
        try
        {
          if(stmt != null) stmt.close();
        }
        catch (java.sql.SQLException sqle)
        {
          //usually can ignore
        }
        try
        {
          if(conn != null) conn.close();
        }
        catch (java.sql.SQLException sqle)
        {
          //usually can ignore
        }
      }
    }
    while (retry);
  }

  public boolean causedByStaleConnection(java.rmi.EJBException re)
  {
    // Search the exception chain for errors
    // indicating a stale connection
    for (Throwable t = re; t != null; t = t.getCause())
      if (t instanceof RetryableConnectionException)
        return true;

    // Not found to be stale
    return false;
  }
}

public class MyEntityBean extends javax.ejb.EntityBean
{
  ... other methods here ...
  public void ejbStore() throws java.rmi.EJBException
  {
    try
    {
      conn = ds.getConnection();
      stmt = conn.createStatement();
      stmt.execute("UPDATE my_table SET value=1 WHERE
      primaryKey=" + myPrimaryKey);
    }
    catch(java.sql.SQLException sqlX)
    {
      // Find out if the error indicates the connection is stale
      if (com.ibm.websphere.rsadapter.WSCallHelper
          .getDataStoreHelper(ds)
          .isConnectionError(sqlX))
      {
        // rollback the tran when method returns
        getEntityContext().setRollbackOnly();
        throw new java.rmi.EJBException(
          "Exception occurred in ejbStore",
          new RetryableConnectionException(sqlX));
      }
      else
      {
        // handle some other problem
      }
    }
    finally
    {
```

```
      //always cleanup JDBC resources
      try
      {
        if(stmt != null) stmt.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
      try
      {
        if(conn != null) conn.close();
      }
      catch (java.sql.SQLException sqle)
      {
        //usually can ignore
      }
    }
  }
}
```

In *mySessionBMTMethod()* of the previous example:

- The session bean first retrieves a *UserTransaction* object from the session context and then begins a global transaction.
- Next, it calls a method on the entity bean, which calls the *ejbLoad()* method. If *ejbLoad()* runs successfully, the client then commits the transaction, causing the *ejbStore()* method to be called.
- In *ejbStore()*, the entity bean gets a connection and writes its state to the database; if the connection retrieved is stale, the transaction is marked *rollbackOnly* and a new *EJBException* that wraps the *RetryableConnectionException* is thrown. That exception is then caught by the client, which cleans up the JDBC resources, rolls back the transaction, and calls *causedByStaleConnection()*, which determines if a stale connection exception is buried somewhere in the exception.
- If the method returns true, the retry flag is set and the transaction is retried; otherwise, the exception is re-issued to the caller.
- The *causedByStaleConnection()* method looks through the chain of detail attributes to find the original exception. Multiple wrapping of exceptions can occur by the time the exception finally gets back to the client, so the method keeps searching until it encounters stale connection exception and `true` is returned; otherwise, there is no stale connection exception in the list and `false` is returned.
- If you are talking to a CMP bean instead of to a BMP bean, the session bean is the same. The CMP bean *ejbStore()* method would most likely be empty, and the container after calling it would persist the bean with generated code.
- If a stale connection exception occurs during persistence, it is wrapped with a remote exception and returned to the caller. The *causedByStaleConnection()* method would again look through the exception chain and find the root exception, which would be stale connection exception.

Objects operating in a local transaction context

When a database operation occurs outside of a global transaction context, a local transaction is implicitly begun by the container. This includes servlets or JSPs that do not begin transactions with the *UserTransaction* interface, as well as enterprise beans running in unspecified transaction contexts. As with global transactions, you must roll back the local transaction before the operation is retried. In these cases, the local transaction containment usually ends when the business method ends. The one exception is if you are using activity sessions. In this case the activity session must end before attempting to get a new connection.

When the local transaction occurs in an enterprise bean running in an unspecified transaction context, the enterprise bean client object, outside of the local transaction containment, could use the method described in the previous bullet to retry the transaction. However, when the local transaction containment takes place as part of a servlet or JSP file, there is no client object available to retry the operation. For this reason, it is recommended to avoid database operations in servlets and JSP files unless they are a part of a user transaction.

Stale Connection on Linux systems

You might must set a loopback to access DB2 databases from the application server on a Linux platform.

A Linux semaphore issue can interfere with JDBC access to your DB2 database in either of these configurations:

- Using the DB2 Universal JDBC Type 2 driver to connect to a local DB2 database
- Using the DB2 Universal JDBC Type 2 driver to access DB2 for z/OS through a DB2 Connect™ installation on the same machine as the application server. The problem occurs only if DB2 Connect restricts local clients from running within an agent. (That is, if the DB2_IN_APP_PROCESS setting is not the default value, or if the setting is `Yes`. Set the value to `No` to fix the problem and avoid performing the following procedure.)

The issue often triggers the JVM logs to display the DB2 stale connection exception `SQL1224`. Because the SQL exception code can vary, however, check the DB2 trace log when you encounter a stale connection. If you see the following error data, the Linux semaphore behavior is the problem:

```
'71' -SQLCC_ERR_CONN_CLOSED_BY_PARTNER and SQLCODE -XXXX
```

To work around the problem, set the loopback for your database. For example, if your database name is `WAS`, host name is `LHOST`, and database service port number is `50000`, issue the following commands from the DB2 command-line window:

```
db2 catalog TCPIP node RHOST remote LHOST server 50000
db2 uncatalog db WAS
db2 catalog db WAS as WASAlias at node loop authentication server
//If you connect to WASAlias, it is connect through loopback;
//If you connect to WAS, it is "normal" connect.
db2 catalog db WASAlias as WAS at node RHOST
```

Example: Handling servlet JDBC connection exceptions

The following code sample demonstrates how to set transaction management and connection management properties, such as operation retries, to address stale connection exceptions within a servlet JDBC transaction.

This example code performs the following actions:

- initializes a servlet
- looks up a data source
- specifies error messages, connection retries, and transaction rollback requirements

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.util.*;
// Import JDBC packages and naming service packages.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.transaction.*;
import com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException;
import com.ibm.websphere.rsadapter.WSCallHelper;

public class EmployeeListTran extends HttpServlet {
    private static DataSource ds   = null;
    private UserTransaction ut = null;
    private static String title    = "Employee List";

// ****************************************************************
// * Initialize servlet when it is first loaded.                  *
// * Get information from the properties file, and look up the    *
// * DataSource object from JNDI to improve performance of the    *
// * the servlet's service methods.                               *
// ****************************************************************
    public void init(ServletConfig config)
    throws ServletException
    {
        super.init(config);
        getDS();
    }

// ****************************************************************
// * Perform the JNDI lookup for the DataSource and               *
// * User Transaction objects.                                    *
// * This method is invoked from init(), and from the service     *
// * method of the DataSource is null                             *
// ****************************************************************
    private void getDS() {
        try {
            Hashtable parms = new Hashtable();
            parms.put(Context.INITIAL_CONTEXT_FACTORY,
 com.ibm.websphere.naming.WsnInitialContextFactory);
            InitialContext ctx = new InitialContext(parms);
            // Perform a naming service lookup to get the DataSource object.
            ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
            ut = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
        } catch (Exception e) {
            System.out.println("Naming service exception:" + e.getMessage());
            e.printStackTrace();
        }
    }

// ****************************************************************
// * Respond to user GET request                                  *
// ****************************************************************
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException
    {
        Connection conn  = null;
        Statement stmt = null;
        ResultSet rs = null;
        Vector employeeList = new Vector();
        // Set retryCount to the number of times you would like to retry after a
        // stale connection exception
        int retryCount = 5;
        // If the Database code processes successfully, we will set error = false
        boolean error = true;
        do
        {
            try
            {
                //Start a new Transaction
                ut.begin();
                // Get a Connection object conn using the DataSource factory.
                conn = ds.getConnection();
                // Run DB query using standard JDBC coding.
```

```
                stmt = conn.createStatement();
                String query   = "Select FirstNme, MidInit, LastName" +
                                 "from Employee ORDER BY LastName";
                rs   = stmt.executeQuery(query);
                while (rs.next())
                {
                    employeeList.addElement(rs.getString(3) + ", "+ rs.getString(1)" +     " + rs.getString(2));
                }
                //Set error to false to indicate successful completion of the database work
                error=false;
            }
            catch (SQLException sqlX)
            {
              // Determine if the connection request timed out.
              // This code works regardless of which error detection
              // model is used. If exception mapping is enabled, then
              // we need to look for ConnectionWaitTimeoutException.
              // If exception checking is enabled, then look for
              // SQLTransientConnectionException with a chained
              // ConnectionWaitTimeoutException.

              if ( sqlX instanceof ConnectionWaitTimeoutException
                || sqlX instanceof SQLTransientConnectionException
                   && sqlX.getCause() instanceof ConnectionWaitTimeoutException)
              {
                // This exception is thrown if a connection can not be obtained from the
                // pool within a configurable amount of time.  Frequent occurrences of
                // this exception indicate an incorrectly tuned connection pool

                System.out.println("Connection Wait Timeout Exception during get connection or
process SQL:" + c.getMessage());

                //In general, we do not want to retry after this exception, so set retry count to 0
                //and roll back the transaction
                try
                {
                    ut.setRollbackOnly();
                }
                catch (SecurityException se)
                {
                    //Thrown to indicate that the thread is not allowed to roll back the transaction.
                    System.out.println("Security Exception setting rollback only!" + se.getMessage());
                }
                catch (IllegalStateException ise)
                {
                    //Thrown if the current thread is not associated with a transaction.
                    System.out.println("Illegal State Exception setting rollback only!" + ise.getMessage());
                }
                catch (SystemException sye)
                {
                    //Thrown if the transaction manager encounters an unexpected error condition
                    System.out.println("System Exception setting rollback only!" + sye.getMessage());
                }
                retryCount=0;
              }
              else if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(sqlX))
              {
                // This exception indicates that the connection to the database is no longer valid.
                //Roll back the transaction, then retry several times to attempt to obtain a valid
                //connection, display an error message if the connection still can not be obtained.

                System.out.println("Connection is stale:" + sc.getMessage());

                try
                {
                    ut.setRollbackOnly();
                }
                catch (SecurityException se)
                {
                    //Thrown to indicate that the thread is not allowed to roll back the transaction.
                    System.out.println("Security Exception setting rollback only!" + se.getMessage());
                }
```

```
    catch (IllegalStateException ise)
    {
        //Thrown if the current thread is not associated with a transaction.
        System.out.println("Illegal State Exception setting rollback only!" + ise.getMessage());
    }
    catch (SystemException sye)
    {
        //Thrown if the transaction manager encounters an unexpected error condition
        System.out.println("System Exception setting rollback only!" + sye.getMessage());
    }
    if (--retryCount == 0)
    {
        System.out.println("Five stale connection exceptions, displaying error page.");
    }
  }
  else
  {
    System.out.println("SQL Exception during get connection or process SQL: " + sq.getMessage());

    //In general, we do not want to retry after this exception, so set retry count to 0
    //and rollback the transaction
    try
    {
        ut.setRollbackOnly();
    }
    catch (SecurityException se)
    {
        //Thrown to indicate that the thread is not allowed to roll back the transaction.
        System.out.println("Security Exception setting rollback only!" + se.getMessage());
    }
    catch (IllegalStateException ise)
    {
        //Thrown if the current thread is not associated with a transaction.
        System.out.println("Illegal State Exception setting rollback only!" + ise.getMessage());
    }
    catch (SystemException sye)
    {
        //Thrown if the transaction manager encounters an unexpected error condition
        System.out.println("System Exception setting rollback only!" + sye.getMessage());
    }
    retryCount=0;
  }
}
catch (NotSupportedException nse)
{
    //Thrown by UserTransaction begin method if the thread is already associated with a
    //transaction and the Transaction Manager implementation does not support nested
    //transactions.
    System.out.println("NotSupportedException on User Transaction begin:" + nse.getMessage());
}
catch (SystemException se)
{
    //Thrown if the transaction manager encounters an unexpected error condition
    System.out.println("SystemException in User Transaction:" +se.getMessage());
}
catch (Exception e)
{
    System.out.println("Exception in get connection or process SQL:" + e.getMessage());
    //In general, we do not want to retry after this exception, so set retry count to 5
    //and roll back the transaction
    try
    {
        ut.setRollbackOnly();
    }
    catch (SecurityException se)
    {
        //Thrown to indicate that the thread is not allowed to roll back the transaction.
        System.out.println("Security Exception setting rollback only!" + se.getMessage());
    }
    catch (IllegalStateException ise)
    {
        //Thrown if the current thread is not associated with a transaction.
```

```
            System.out.println("Illegal State Exception setting rollback only!" + ise.getMessage());
        }
        catch (SystemException sye)
        {
            //Thrown if the transaction manager encounters an unexpected error condition
            System.out.println("System Exception setting rollback only!" + sye.getMessage());
        }
        retryCount=0;
    }
    finally
    {
        // Always close the connection in a finally statement to ensure proper
        // closure in all cases. Closing the connection does not close and
        // actual connection, but releases it back to the pool for reuse.

        if (rs != null)
        {
           try
           {
              rs.close();
           }
           catch (Exception e)
           {
              System.out.println("Close Resultset Exception:" + e.getMessage());
           }
        }
        if (stmt != null)
        {
           try
           {
              stmt.close();
           }
           catch (Exception e)
           {
              System.out.println("Close Statement Exception:" + e.getMessage());
           }
        }
        if (conn != null)
        {
           try
           {
              conn.close();
           }
           catch (Exception e)
           {
              System.out.println("Close connection exception:" + e.getMessage());
           }
        }
        try
        {
            ut.commit();
        }
        catch (RollbackException re)
        {
            //Thrown to indicate that the transaction has been rolled back rather than committed.
            System.out.println("User Transaction Rolled back!" + re.getMessage());
        }
        catch (SecurityException se)
        {
            //Thrown to indicate that the thread is not allowed to commit the transaction.
            System.out.println("Security Exception thrown on transaction commit:" + se.getMessage());
        }
        catch (IllegalStateException ise)
        {
            //Thrown if the current thread is not associated with a transaction.
            System.out.println("Illegal State Exception thrown on transaction commit:" + ise.getMessage());
        }
        catch (SystemException sye)
        {
            //Thrown if the transaction manager encounters an unexpected error condition
            System.out.println("System Exception thrown on transaction commit:" + sye.getMessage());
        }
```

```
                catch (Exception e)
                {
                    System.out.println("Exception thrown on transaction commit:" + e.getMessage());
                }
            }
        }
        while ( error==true && retryCount > 0 );

        // Prepare and return HTML response, prevent dynamic content from being cached
        // on browsers.
        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setHeader("Cache-Control", "no-cache");
        res.setDateHeader("Expires", 0);
        try
        {
            ServletOutputStream out = res.getOutputStream();
            out.println("<HTML>");
            out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>");
            out.println("<BODY>");
            if (error==true)
            {
                out.println("<H1>There was an error processing this request.</H1>" +
                  "Please try the request again, or contact" +
                   "the <a href='mailto:sysadmin@my.com'>System Administrator</a>");
            }
            else if (employeeList.isEmpty())
            {
               out.println("<H1>Employee List is Empty</H1>");
            }
            else
            {
               out.println("<H1>Employee List </H1>");
               for (int i = 0; i < employeeList.size(); i++)
               {
                  out.println(employeeList.elementAt(i) + "<BR>");
               }
            }
            out.println("</BODY></HTML>");
            out.close();
        }
        catch (IOException e)
        {
           System.out.println("HTML response exception:" + e.getMessage());
        }
    }
}
```

Example: Handling connection exceptions for session beans in container-managed database transactions

The following code sample demonstrates how to roll back transactions and issue exceptions to the bean client in cases of stale connection exceptions.

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
```

```
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException;
import com.ibm.websphere.rsadapter.WSCallHelper;

/*****************************************************************************
* This bean is designed to demonstrate Database Connections in a
* Container Managed Transaction Session Bean.  Its transaction attribute              *
* should be set to TX_REQUIRED or TX_REQUIRES_NEW.                                    *
*****************************************************************************
*/
public class ShowEmployeesCMTBean implements SessionBean {
   private javax.ejb.SessionContext mySessionCtx = null;
   final static long serialVersionUID = 3206093459760846163L;

   private javax.sql.DataSource ds;

//****************************************************************************
//* ejbActivate calls the getDS method, which does the JNDI lookup for the DataSource.
//* Because the DataSource lookup is in a separate method, we can also invoke it from
//* the getEmployees method in the case where the DataSource field is null.
//****************************************************************************
public void ejbActivate() throws java.rmi.EJBException {
   getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.EJBException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.EJBException {}
/**
 * ejbPassivate method
 * @exception java.rmi.EJBException
 */
public void ejbPassivate() throws java.rmi.EJBException {}
/**
 * ejbRemove method
 * @exception java.rmi.EJBException
 */
public void ejbRemove() throws java.rmi.EJBException {}

//****************************************************************************
//* The getEmployees method runs the database query to retrieve the employees.
//* The getDS method is only called if the DataSource variable is null.
//* Because this session bean uses Container Managed Transactions, it cannot retry the
//* transaction on a StaleConnectionException.  However, it can throw an exception to
//* its client indicating that the operation is retriable.
//****************************************************************************

public Vector getEmployees() throws ConnectionWaitTimeoutException, SQLException,
       RetryableConnectionException
{
   Connection conn  = null;
   Statement stmt = null;
   ResultSet rs = null;
   Vector employeeList = new Vector();

   if (ds == null) getDS();

   try
   {
      // Get a Connection object conn using the DataSource factory.
      conn = ds.getConnection();
      // Run DB query using standard JDBC coding.
```

```
   stmt = conn.createStatement();
   String query   = "Select FirstNme, MidInit, LastName" +
                    "from Employee ORDER BY LastName;"
   rs   = stmt.executeQuery(query);
   while (rs.next())
   {
      employeeList.addElement(rs.getString(3) + ", "+ rs.getString(1)" + " + rs.getString(2));
   }
}
catch (SQLException sqlX)
{
  // Determine if the connection is stale
  if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(sqlX))
  {
    // This exception indicates that the connection to the database is no longer valid.
    // Roll back the transaction, and throw an exception to the client indicating they
    // can retry the transaction if desired.

    System.out.println("Connection is stale:" + sqlX.getMessage());
    System.out.println("Rolling back transaction and throwing  RetryableConnectionException");

    mySessionCtx.setRollbackOnly();
    throw new RetryableConnectionException(sqlX.toString());
  }
  // Determine if the connection request timed out.
  else if ( sqlX instanceof ConnectionWaitTimeoutException
         || sqlX instanceof SQLTransientConnectionException
            && sqlX.getCause() instanceof ConnectionWaitTimeoutException)
  {
    // This exception is thrown if a connection can not be obtained from the
    // pool within a configurable amount of time.  Frequent occurrences of
    // this exception indicate an incorrectly tuned connection pool

    System.out.println("Connection Wait Timeout Exception during get connection or process SQL:" +
    sqlX.getMessage());
    throw sqlX instanceof ConnectionWaitTimeoutException ?
          sqlX :
          (ConnectionWaitTimeoutException) sqlX.getCause();
  }
  else
  {
    //Throwing a remote exception will automatically roll back the container managed
    //transaction

    System.out.println("SQL Exception during get connection or process SQL:" +
         sqlX.getMessage());
    throw sqlX;
  }
}
finally
{
  // Always close the connection in a finally statement to ensure proper
  // closure in all cases. Closing the connection does not close and
  // actual connection, but releases it back to the pool for reuse.

  if (rs != null)
  {
    try
    {
      rs.close();
    }
    catch (Exception e)
    {
      System.out.println("Close Resultset Exception:" +
                    e.getMessage());
    }
  }
  if (stmt != null)
  {
    try
    {
      stmt.close();
```

```
      }
      catch (Exception e)
      {
        System.out.println("Close Statement Exception:" +
                    e.getMessage());
      }
    }
    if (conn != null)
    {
      try
      {
        conn.close();
      }
      catch (Exception e)
      {
        System.out.println("Close connection exception:" + e.getMessage());
      }
    }
  }
  return employeeList;
}

/**
 * getSessionContext method
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
  return mySessionCtx;
}
//*****************************************************************************
//* The getDS method performs the JNDI lookup for the data source.
//* This method is called from ejbActivate, and from getEmployees if the data source
//* object is null.
//*****************************************************************************

private void getDS() {
  try {
     Hashtable parms = new Hashtable();
     parms.put(Context.INITIAL_CONTEXT_FACTORY,
           com.ibm.websphere.naming.WsnInitialContextFactory);
     InitialContext ctx = new InitialContext(parms);
     // Perform a naming service lookup to get the DataSource object.
     ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
  }
      catch (Exception e) {
     System.out.println("Naming service exception:" + e.getMessage());
     e.printStackTrace();
  }
}
/**
 * setSessionContext method
 * @param ctx javax.ejb.SessionContext
 * @exception java.rmi.EJBException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws java.rmi.EJBException {
  mySessionCtx = ctx;
}
}
//==================START_PROLOG=====================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
```

```
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Session Bean
 */
public interface ShowEmployeesCMTHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
* @return WebSphereSamples.ConnPool.ShowEmployeesCMT
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
WebSphereSamples.ConnPool.ShowEmployeesCMT create() throws javax.ejb.CreateException,
   java.rmi.RemoteException;
}
//==================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface ShowEmployeesCMT extends javax.ejb.EJBObject {

/**
 *
 * @return java.util.Vector
 */
java.util.Vector getEmployees() throws java.sql.SQLException, java.rmi.RemoteException,
 ConnectionWaitTimeoutException, WebSphereSamples.ConnPool.RetryableConnectionException;
}
//==================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
```

```
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * Exception indicating that the operation can be retried
 * Creation date: (4/2/2001 10:48:08 AM)
 * @author: Administrator
 */
public class RetryableConnectionException extends Exception {
/**
 * RetryableConnectionException constructor.
 */
public RetryableConnectionException() {
   super();
}
/**
 * RetryableConnectionException constructor.
 * @param s java.lang.String
 */
public RetryableConnectionException(String s) {
   super(s);
}
}
```

Example: Handling connection exceptions for session beans in bean-managed database transactions

The following code sample demonstrates your options for addressing stale connection exceptions. You can set different transaction management and connection management parameters, such as the number of operation retries, and the connection timeout interval.

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import javax.transaction.*;
import com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException;
import com.ibm.websphere.rsadapter.WSCallHelper;

/*****************************************************************************
* This bean is designed to demonstrate Database Connections in a                *
* Bean-Managed Transaction Session Bean.  Its transaction attribute             *
* should be set to TX_BEANMANAGED.
*****************************************************************************/
public class ShowEmployeesBMTBean implements SessionBean {
   private javax.ejb.SessionContext mySessionCtx = null;
```

```
   final static long serialVersionUID = 3206093459760846163L;

   private javax.sql.DataSource ds;

   private javax.transaction.UserTransaction userTran;

//*****************************************************************************
//* ejbActivate calls the getDS method, which makes the JNDI lookup for the DataSource
//* Because the DataSource lookup is in a separate method, we can also invoke it from
//* the getEmployees method in the case where the DataSource field is null.
//*****************************************************************************
public void ejbActivate() throws java.rmi.EJBException {
   getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.EJBException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.EJBException {}
/**
 * ejbPassivate method
 * @exception java.rmi.EJBException
 */
public void ejbPassivate() throws java.rmi.EJBException {}
/**
 * ejbRemove method
 * @exception java.rmi.EJBException
 */
public void ejbRemove() throws java.rmi.EJBException {}

//*****************************************************************************
//* The getEmployees method runs the database query to retrieve the employees.
//* The getDS method is only called if the DataSource or userTran variables are null.
//* If a stale connection occurs, the bean retries the transaction 5 times,
//* then throws an EJBException.
//*****************************************************************************

public Vector getEmployees() throws EJBException {
   Connection conn  = null;
   Statement stmt = null;
   ResultSet rs = null;
   Vector employeeList = new Vector();

   // Set retryCount to the number of times you would like to retry after a
   // stale connection
   int retryCount = 5;

   // If the Database code processes successfully, we will set error = false
   boolean error = true;

   if (ds == null || userTran == null) getDS();
   do
   {
         try
         {
            //try/catch block for UserTransaction work
            //Begin the transaction
            userTran.begin();
            try
            {
               //try/catch block for database work
               //Get a Connection object conn using the DataSource factory.
               conn = ds.getConnection();
               // Run DB query using standard JDBC coding.
               stmt = conn.createStatement();
               String query   = "Select FirstNme, MidInit, LastName" +
             "from Employee ORDER BY LastName";
               rs   = stmt.executeQuery(query);
               while (rs.next())
               {
                  employeeList.addElement(rs.getString(3) + ", "+ rs.getString(1) +" " + rs.getString(2));
```

```
            }
            //Set error to false, as all database operations are successfully completed
            error = false;
          }
          catch (SQLException sqlX)
          {
            if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(sqlX))
            {
              // This exception indicates that the connection to the database is no longer valid.
              // Rollback the transaction, and throw an exception to the client indicating they
              // can retry the transaction if desired.

              System.out.println("Stale connection:" +
              se.getMessage());
              userTran.rollback();
              if (--retryCount == 0)
              {
                //If we have already retried the requested number of times, throw an EJBException.
                throw new EJBException("Transaction Failure:" + sqlX.toString());
              }
              else
              {
                System.out.println("Retrying transaction, retryCount =" +
                       retryCount);
              }
            }
            else if (sqlX instanceof ConnectionWaitTimeoutException
                  || sqlX instanceof SQLTransientConnectionException
                     && sqlX.getCause() instanceof ConnectionWaitTimeoutException)
            {
              // This exception is thrown if a connection can not be obtained from the
              // pool within a configurable amount of time.  Frequent occurrences of
              // this exception indicate an incorrectly tuned connection pool

              System.out.println("Connection request timed out:" +
              sqlX.getMessage());
              userTran.rollback();
              throw new EJBException("Transaction failure:" + sqlX.getMessage());
            }
            else
            {
        // This catch handles all other SQL Exceptions
              System.out.println("SQL Exception during get connection or process SQL:" +
          sqlX.getMessage());
              userTran.rollback();
              throw new EJBException("Transaction failure:" + sqlX.getMessage());
            }
          finally
          {
            // Always close the connection in a finally statement to ensure proper
            // closure in all cases. Closing the connection does not close and
            // actual connection, but releases it back to the pool for reuse.

            if (rs != null) {
             try {
                rs.close();
             }
             catch (Exception e) {
                System.out.println("Close Resultset Exception:" + e.getMessage());
             }
          }
          if (stmt != null) {
             try {
                stmt.close();
}
          catch (Exception e) {
             System.out.println("Close Statement Exception:" + e.getMessage());
          }
        }
        if (conn != null) {
          try {
             conn.close();
```

```
            }
            catch (Exception e) {
               System.out.println("Close connection exception:" + e.getMessage());
            }
         }
      }
      if (!error) {
         //Database work completed successfully, commit the transaction
         userTran.commit();
      }
      //Catch UserTransaction exceptions
      }
      catch (NotSupportedException nse) {

//Thrown by UserTransaction begin method if the thread is already associated with a
//transaction and the Transaction Manager implementation does not support nested transactions.
 System.out.println("NotSupportedException on User Transaction begin:" +
                                nse.getMessage());
         throw new EJBException("Transaction failure:" + nse.getMessage());
      }
      catch (RollbackException re) {
//Thrown to indicate that the transaction has been rolled back rather than committed.
         System.out.println("User Transaction Rolled back!" + re.getMessage());
         throw new EJBException("Transaction failure:" + re.getMessage());
      }
      catch (SystemException se) {
     //Thrown if the transaction manager encounters an unexpected error condition
         System.out.println("SystemException in User Transaction:" + se.getMessage());
         throw new EJBException("Transaction failure:" + se.getMessage());
      }
      catch (Exception e) {
         //Handle any generic or unexpected Exceptions
         System.out.println("Exception in User Transaction:" + e.getMessage());
         throw new EJBException("Transaction failure:" + e.getMessage());
      }
   }
   while (error);
   return employeeList;
}
/**
 * getSessionContext method comment
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
   return mySessionCtx;
}

//*******************************************************************************
//* The getDS method performs the JNDI lookup for the DataSource.
//* This method is called from ejbActivate, and from getEmployees if the DataSource
//* object is null.
//*******************************************************************************
private void getDS() {
   try {
      Hashtable parms = new Hashtable();
parms.put(Context.INITIAL_CONTEXT_FACTORY,
         com.ibm.websphere.naming.WsnInitialContextFactory);
      InitialContext ctx = new InitialContext(parms);

      // Perform a naming service lookup to get the DataSource object.
      ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
      //Create the UserTransaction object
      userTran = mySessionCtx.getUserTransaction();
   }
   catch (Exception e) {
      System.out.println("Naming service exception:" + e.getMessage());
      e.printStackTrace();
}
}
/**
 * setSessionContext method
 * @param ctx javax.ejb.SessionContext
```

```
 * @exception java.rmi.EJBException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws java.rmi.EJBException {
   mySessionCtx = ctx;
}
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Session Bean
 */
public interface ShowEmployeesBMTHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
 * @return WebSphereSamples.ConnPool.ShowEmployeesBMT
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
WebSphereSamples.ConnPool.ShowEmployeesBMT create() throws javax.ejb.CreateException,
    java.rmi.RemoteException;
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface ShowEmployeesBMT extends javax.ejb.EJBObject {

/**
 *
```

```
 * @return java.util.Vector
 */
java.util.Vector getEmployees() throws java.rmi.RemoteException, javax.ejb.EJBException;
}
```

Example: Handling connection exceptions for BMP beans in container-managed database transactions

The following code sample demonstrates how to roll back transactions and issue exceptions to the bean client in cases of stale connection exceptions.

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2005,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import com.ibm.websphere.rsadapter.WSCallHelper;

/**
 * This is an Entity Bean class with five BMP fields
 * String firstName, String lastName, String middleInit
 * String empNo, int edLevel
 */
public class EmployeeBMPBean implements EntityBean {
       private javax.ejb.EntityContext entityContext = null;
       final static long serialVersionUID = 3206093459760846163L;

       private java.lang.String firstName;
       private java.lang.String lastName;
       private String middleInit;
       private javax.sql.DataSource ds;
       private java.lang.String empNo;
       private int edLevel;
/**
 * ejbActivate method
 * ejbActivate calls getDS(), which performs the
 * JNDI lookup for the datasource.
 */
public void ejbActivate() {
       getDS();
}
/**
 * ejbCreate method for a BMP entity bean
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception javax.ejb.CreateException
 */
public WebSphereSamples.ConnPool.EmployeeBMPKey ejbCreate(String empNo,
String firstName, String lastName, String middleInit, int edLevel) throws
```

```
javax.ejb.CreateException {

   Connection conn = null;
   PreparedStatement ps = null;

   if (ds == null) getDS();

   this.empNo = empNo;
   this.firstName = firstName;
   this.lastName = lastName;
   this.middleInit = middleInit;
   this.edLevel = edLevel;

   String sql = "insert into Employee (empNo, firstnme, midinit, lastname,
        edlevel) values (?,?,?,?,?)";

   try {
      conn = ds.getConnection();
      ps = conn.prepareStatement(sql);
      ps.setString(1, empNo);
      ps.setString(2, firstName);
      ps.setString(3, middleInit);
      ps.setString(4, lastName);
      ps.setInt(5, edLevel);

 if (ps.executeUpdate() != 1){
         System.out.println("ejbCreate Failed to add user.");
         throw new CreateException("Failed to add user.");
      }
   }
   catch (SQLException se)
   {
      if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(se))
      {
         // This exception indicates that the connection to the database is no longer valid.
         // Rollback the transaction, and throw an exception to the client indicating they
         // can retry the transaction if desired.

         System.out.println("Connection is stale:" +  se.getMessage());
         throw new CreateException(se.getMessage());
      }
      else
      {
         System.out.println("SQL Exception during get connection or process SQL:" +
                           se.getMessage());
         throw new CreateException(se.getMessage());
      }
   }
   finally
   {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close an
      // actual connection, but releases it back to the pool for reuse.
      if (ps != null)
      {
         try
         {
            ps.close();
         }
         catch (Exception e)
         {
            System.out.println("Close Statement Exception:" + e.getMessage());
         }
      }
      if (conn != null)
      {
         try
         {
            conn.close();
         }
         catch (Exception e)
         {
```

```
            System.out.println("Close connection exception:" + e.getMessage());
         }
      }
   }
   return new EmployeeBMPKey(this.empNo);
}
/**
 * ejbFindByPrimaryKey method
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param primaryKey WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception javax.ejb.FinderException
 */
public WebSphereSamples.ConnPool.EmployeeBMPKey
     ejbFindByPrimaryKey(WebSphereSamples.ConnPool.EmployeeBMPKey primaryKey)
                 javax.ejb.FinderException {
       loadByEmpNo(primaryKey.empNo);
       return primaryKey;
}
/**
 * ejbLoad method
 */
public void ejbLoad() {
       try {
              EmployeeBMPKey pk = (EmployeeBMPKey) entityContext.getPrimaryKey();
              loadByEmpNo(pk.empNo);
       } catch (FinderException fe) {
              throw new EJBException("Cannot load Employee state from database.");
       }
}
/**
 * ejbPassivate method
 */
public void ejbPassivate() {}
/**
 * ejbPostCreate method for a BMP entity bean
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 */
public void ejbPostCreate(String empNo, String firstName, String lastName, String middleInit,
    int edLevel) {}
/**
 * ejbRemove method
 * @exception javax.ejb.RemoveException
 */
public void ejbRemove() throws javax.ejb.RemoveException
{

   if (ds == null)
      GetDS();

   String sql = "delete from Employee where empNo=?";
   Connection con = null;
   PreparedStatement ps = null;
   try
   {
      con = ds.getConnection();
      ps = con.prepareStatement(sql);
      ps.setString(1, empNo);
      if (ps.executeUpdate() != 1)
      {
         throw new EJBException("Cannot remove employee:" + empNo);
      }
   }
   catch (SQLException se)
   {
      if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(se))
      {
         // This exception indicates that the connection to the database is no longer valid.
         // Rollback the transaction, and throw an exception to the client indicating they
         // can retry the transaction if desired.

         System.out.println("Connection is stale:" + se.getMessage());
         throw new EJBException(se.getMessage());
```

```
      }
      else
      {
         System.out.println("SQL Exception during get connection or process SQL:" +
                            se.getMessage());
         throw new EJBException(se.getMessage());
      }
   }
   finally
   {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close an
      // actual connection, but releases it back to the pool for reuse.
      if (ps != null)
      {
         try
         {
            ps.close();
         }
         catch (Exception e)
         {
            System.out.println("Close Statement Exception:" + e.getMessage());
         }
      }
      if (con != null)
      {
         try
         {
            con.close();
         }
         catch (Exception e)
         {
            System.out.println("Close connection exception:" + e.getMessage());
         }
      }
   }
}
/**
 * Get the employee's edLevel
 * Creation date: (4/20/2001 3:46:22 PM)
 * @return int
 */
public int getEdLevel() {
       return edLevel;
}
/**
 * getEntityContext method
 * @return javax.ejb.EntityContext
 */
public javax.ejb.EntityContext getEntityContext() {
       return entityContext;
}
/**
 * Get the employee's first name
 * Creation date: (4/19/2001 1:34:47 PM)
 * @return java.lang.String
 */
public java.lang.String getFirstName() {
       return firstName;
}
/**
 * Get the employee's last name
 * Creation date: (4/19/2001 1:35:41 PM)
 * @return java.lang.String
 */
public java.lang.String getLastName() {
       return lastName;
}
/**
* get the employee's middle initial
 * Creation date: (4/19/2001 1:36:15 PM)
 * @return char
```

```
 */
public String getMiddleInit() {
       return middleInit;
}
/**
 * Lookup the DataSource from JNDI
 * Creation date: (4/19/2001 3:28:15 PM)
 */
private void getDS() {
       try {
              Hashtable parms = new Hashtable();
              parms.put(Context.INITIAL_CONTEXT_FACTORY,
                           com.ibm.websphere.naming.WsnInitialContextFactory);
              InitialContext ctx = new InitialContext(parms);
              // Perform a naming service lookup to get the DataSource object.
              ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
       }
       catch (Exception e) {
              System.out.println("Naming service exception:" + e.getMessage());
              e.printStackTrace();
       }
}
/**
 * Load the employee from the database
 * Creation date: (4/19/2001 3:44:07 PM)
 * @param empNo java.lang.String
 */
private void loadByEmpNo(String empNoKey) throws javax.ejb.FinderException
{
       String sql = "select empno, firstnme, midinit, lastname, edLevel from employee where empno = ?";
       Connection conn = null;
       PreparedStatement ps = null;
       ResultSet rs = null;

       if (ds == null) getDS();

       try
       {
              // Get a Connection object conn using the DataSource factory.
              conn = ds.getConnection();
              // Run DB query using standard JDBC coding.
              ps = conn.prepareStatement(sql);
              ps.setString(1, empNoKey);
              rs = ps.executeQuery();
              if (rs.next())
              {
                     empNo= rs.getString(1);
                     firstName=rs.getString(2);
                     middleInit=rs.getString(3);
                     lastName=rs.getString(4);
                     edLevel=rs.getInt(5);
              }
              else
              {
                throw new ObjectNotFoundException("Cannot find employee number" +
                                                  empNoKey);
              }
       }
       catch (SQLException se)
       {
         if (WSCallHelper.getDataStoreHelper(ds).isConnectionError(se))
         {
           // This exception indicates that the connection to the database is no longer valid.
           // Roll back the transaction, and throw an exception to the client indicating they
           // can retry the transaction if desired.

           System.out.println("Connection is stale:" + se.getMessage());
           throw new FinderException(se.getMessage());
         }
         else
         {
           System.out.println("SQL Exception during get connection or process SQL:" +
```

```
                            se.getMessage());
            throw new FinderException(se.getMessage());
          }
        }
        finally
        {
          // Always close the connection in a finally statement to ensure
          // proper closure in all cases. Closing the connection does not
          // close an actual connection, but releases it back to the pool
          // for reuse.
          if (rs != null)
          {
             try
             {
                rs.close();
             }
             catch (Exception e)
             {
                System.out.println("Close Resultset Exception:" + e.getMessage());
             }
          }
          if (ps != null)
          {
             try
             {
                ps.close();
             }
             catch (Exception e)
             {
                System.out.println("Close Statement Exception:" + e.getMessage());
             }
          }
          if (conn != null)
          {
             try
             {
                conn.close();
             }
             catch (Exception e)
             {
                System.out.println("Close connection exception:" + e.getMessage());
             }
          }
        }
}
/**
 * set the employee's education level
 * Creation date: (4/20/2001 3:46:22 PM)
 * @param newEdLevel int
 */
public void setEdLevel(int newEdLevel) {
        edLevel = newEdLevel;
}
/**
 * setEntityContext method
 * @param ctx javax.ejb.EntityContext
 */
public void setEntityContext(javax.ejb.EntityContext ctx) {
        entityContext = ctx;
}
/**
 * set the employee's first name
 * Creation date: (4/19/2001 1:34:47 PM)
 * @param newFirstName java.lang.String
 */
public void setFirstName(java.lang.String newFirstName) {
        firstName = newFirstName;
}
/**
 * set the employee's last name
 * Creation date: (4/19/2001 1:35:41 PM)
 * @param newLastName java.lang.String
```

```
 */
public void setLastName(java.lang.String newLastName) {
       lastName = newLastName;
}
/**
 * set the employee's middle initial
 * Creation date: (4/19/2001 1:36:15 PM)
 * @param newMiddleInit char
 */
public void setMiddleInit(String newMiddleInit) {
       middleInit = newMiddleInit;
}
/**
 * unsetEntityContext method
 */
public void unsetEntityContext() {
       entityContext = null;
}
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface EmployeeBMP extends javax.ejb.EJBObject {

/**
 *
 * @return int
  */
int getEdLevel() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getFirstName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getLastName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getMiddleInit() throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newEdLevel int
 */
void setEdLevel(int newEdLevel) throws java.rmi.RemoteException;
```

```
/**
 *
 * @return void
 * @param newFirstName java.lang.String
 */
void setFirstName(java.lang.String newFirstName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newLastName java.lang.String
 */
void setLastName(java.lang.String newLastName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newMiddleInit java.lang.String
 */
void setMiddleInit(java.lang.String newMiddleInit) throws java.rmi.RemoteException;
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface EmployeeBMP extends javax.ejb.EJBObject {

/**
 *
 * @return int
  */
int getEdLevel() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getFirstName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getLastName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getMiddleInit() throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newEdLevel int
 */
void setEdLevel(int newEdLevel) throws java.rmi.RemoteException;
```

```
/**
 *
 * @return void
 * @param newFirstName java.lang.String
 */
void setFirstName(java.lang.String newFirstName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newLastName java.lang.String
 */
void setLastName(java.lang.String newLastName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newMiddleInit java.lang.String
 */
void setMiddleInit(java.lang.String newMiddleInit) throws java.rmi.RemoteException;
}
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 2002,2008
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
* This is a Primary Key Class for the Entity Bean
**/
public class EmployeeBMPKey implements java.io.Serializable {
   public String empNo;
   final static long serialVersionUID = 3206093459760846163L;

/**
* EmployeeBMPKey() constructor
*/
public EmployeeBMPKey()  {
}
/**
* EmployeeBMPKey(String key) constructor
*/
public EmployeeBMPKey(String key)  {
   empNo = key;
}
/**
* equals method
* - user must provide a proper implementation for the equal method. The generated
*   method assumes the key is a String object.
*/
public boolean equals (Object o)  {
   if (o instanceof EmployeeBMPKey)
      return empNo.equals(((EmployeeBMPKey)o).empNo);
   else
      return false;
}
/**
* hashCode method
```

```
* - user must provide a proper implementation for the hashCode method. The generated
*    method assumes the key is a String object.
*/
public int hashCode ()  {
   return empNo.hashCode();
```

Example: Handling data access exception - ConnectionWaitTimeoutException (for the JDBC API)

This code sample demonstrates how you specify the conditions under which the application server issues the ConnectionWaitTimeoutException for a JDBC application.

In all cases in which the ConnectionWaitTimeoutException is caught, there is little that can be done to recover.

```
public void test1() {
      java.sql.Connection conn = null;
      java.sql.Statement stmt = null;
      java.sql.ResultSet rs = null;

      try {
         // Look for datasource
         java.util.Properties props = new java.util.Properties();
         props.put(
            javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            com.ibm.websphere.naming.WsnInitialContextFactory);
         ic = new javax.naming.InitialContext(props);
         javax.sql.DataSource ds1 = (javax.sql.DataSource) ic.lookup(jndiString);

         // Get Connection.
         conn = ds1.getConnection();
         stmt = conn.createStatement();
         rs = stmt.executeQuery("select * from mytable where this = 54");
      }
      catch (java.sql.SQLException sqlX) {
     if (sqlX instanceof com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException
      || sqlX instanceof java.sql.SQLTransientConnectionException
         && sqlX.getCause() instanceof com.ibm.websphere.ce.cm.ConnectionWaitTimeoutException)
     {
         //notify the user that the system could not provide a
         //connection to the database.  This usually happens when the
         //connection pool is full and there is no connection
         //available for to share.
      }
     else
     {
         // handle other database problems.
     }
      }
      finally {
         if (rs != null)
            try {
               rs.close();
            }
            catch (java.sql.SQLException sqle1) {
            }
         if (stmt != null)
            try {
               stmt.close();
            }
            catch (java.sql.SQLException sqle1) {
            }
         if (conn != null)
            try {
               conn.close();
            }
            catch (java.sql.SQLException sqle1) {
            }
      }
   }
```

Example: Handling data access exception - ConnectionWaitTimeoutException for Java EE Connector Architecture

This code sample demonstrates how you specify the conditions under which WebSphere Application Server issues the ConnectionWaitTimeout exception for a JCA application.

In all cases in which the `ConnectionWaitTimeout` exception is caught, there is little to do for recovery.

The following code fragment shows how to use this exception in Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA):

```
/**
 * This method does a simple Connection test.
 */
public void testConnection()
   throws javax.naming.NamingException, javax.resource.ResourceException,
       com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException {
   javax.resource.cci.ConnectionFactory factory = null;
   javax.resource.cci.Connection conn = null;
   javax.resource.cci.ConnectionMetaData metaData = null;
   try {
      // lookup the connection factory
      if (verbose) System.out.println("Look up the connection factory...");
try {
factory =
  (javax.resource.cci.ConnectionFactory) (new InitialContext()).lookup("java:comp/env/eis/Sample");
  }
   catch (javax.naming.NamingException ne) {
      // Connection factory cannot be looked up.
      throw ne;
   }
    // Get connection
      if (verbose) System.out.println("Get the connection...");
      conn = factory.getConnection();
      // Get ConnectionMetaData
      metaData = conn.getMetaData();
      // Print out the metadata Information.
      System.out.println("EISProductName" is + metaData.getEISProductName());
   }
   catch (com.ibm.websphere.ce.j2c.ConnectionWaitTimeoutException cwtoe) {
      // Connection Wait Timeout
      throw cwtoe;
   }
   catch (javax.resource.ResourceException re) {
      // Something wrong with connections.
      throw re;
   }
   finally {
      if (conn != null) {
         try {
            conn.close();
         }
         catch (javax.resource.ResourceException re) {
         }
      }
   }
}
```

Example: Handling data access exception - error mapping in DataStoreHelper

The application server provides a DataStoreHelper interface for mapping different database SQL error codes to the appropriate exceptions in the application server.

Error mapping is necessary because various database vendors can provide different SQL errors and codes that represent that same issue. For example, the stale connection exception has different codes in

different databases. The DB2 SQLCODEs of 1015, 1034, 1036 , and so on indicate that the connection is no longer available because of a temporary database problem. The Oracle SQLCODEs of 28, 3113, 3114, and so on, indicate the same situation.

Mapping these error codes to standard exceptions provides the consistency that makes applications portable across different installations of the application server. The following code segment illustrates how to add two error codes into the error map:

```
public class NewDSHelper extends GenericDataStoreHelper
{
  public NewDSHelper(java.util.Properties dataStoreHelperProperties)
  {
    super(dataStoreHelperProperties);
    java.util.Hashtable myErrorMap = null;
    myErrorMap = new java.util.Hashtable();
    myErrorMap.put(new Integer(-803), myDuplicateKeyException.class);
    myErrorMap.put(new Integer(-1015), myStaleConnectionException.class);
    myErrorMap.put("S1000", MyTableNotFoundException.class);
    setUserDefinedMap(myErrorMap);
    ...
  }
}
```

A configuration option known as the Error Detection Model controls how the error map is used. At V6 and earlier, Exception Mapping was the only option available for the Error Detection Model. At V7 and later, another option called Exception Checking is also available. Under the Exception Mapping model, the application server consults the error map and replaces exceptions with the corresponding exception type listed in the error map. Under the Exception Checking model, the application server still consults the error map for its own purposes but does not replace exceptions. If you want to continue to use Exception Mapping, you do not need to change anything. Exception Mapping is the default Error Detection Model. If you want to use the Exception Checking Model, see the topic “Changing the Error Detection Model to use the Exception Checking Model” in the related links.

Database deadlock and foreign key conflicts

Repetition of certain SQL error messages indicates problems, such as database referential integrity violations, that you can prevent by using the container managed persistence (CMP) sequence grouping feature.

Exceptions resulting from foreign key conflicts due to violations of database referential integrity

A database referential integrity (RI) policy prescribes rules for how data is written to and deleted from the database tables to maintain relational consistency. Runtime requirements for managing bean persistence, however, can cause an enterprise JavaBeans (EJB) application to violate RI rules, which can cause database exceptions.

Your EJB application is violating database RI if you see an exception message in your WebSphere Application Server trace or log file that is similar to one of the following messages (which were produced in an environment running DB2):

```
The insert or update value of the FOREIGN KEY table1.name_of_foreign_key_constraint
is not equal to any value of the parent key of the parent table.
```

or

```
A parent row cannot be deleted because the relationship table1.name_of_foreign_key_constraint
is not equal to any value of the parent key of the parent table.
```

To prevent these exceptions, you must designate the order in which entity beans update relational database tables by defining sequence groups for the beans.

Exceptions resulting from deadlock caused by optimistic concurrency control schemes

Additionally, sequence grouping can minimize transaction rollback exceptions for entity beans that are configured for optimistic concurrency control. Optimistic concurrency control dictates that database locks be held for minimal amounts of time, so that a maximum number of transactions consistently have access to the data. In such a highly available database, concurrent transactions can attempt to lock the same table row and create deadlock. The resulting exceptions can generate messages similar to the following (which was produced in an environment running DB2):

```
Unsuccessful execution caused by deadlock or timeout.
```

Use the sequence grouping feature to order bean persistence so that database deadlock is less likely to occur.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

  The corresponding product variable is *smpe.install.root*.

  The default is /usr/lpp/zWebSphere/V8R5.

Assembling data access applications

When you assemble enterprise bean code into files that can be deployed onto an application server, you configure properties that define how the application accesses an enterprise information system (EIS), such as a database.

Before you begin

This topic assumes that you have created an enterprise application containing an EJB module that must transact with a database.

About this task

A data access application uses resources, such as data sources or connection factories, to connect with a database.

An application component uses a *connection factory* to access a connection instance, which the component then uses to connect to the underlying enterprise information system (EIS). Examples of connections include database connections, Java Message Service connections, and SAP R/3 connections.

During application assembly you perform activities that enable the application to use these resources. The process typically requires an assembly tool.

Procedure

1. Identify the logical names that are used by the EJB module to reference application resources. These logical names are called *resource references*.

   For further explanation, read the topic, The benefits of using resource references.
2. Start an assembly tool.
3. If you have not done so already, configure the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules. Ensure that **Java EE** capability is enabled.
4. Define mapping and security properties for the resource references. This process includes the following activities:
   a. Bind the resource references to the application resources that provide database connectivity.

      See the topic, Data source lookups for enterprise beans and web modules, for more information on the concept of binding. At deployment time you can alter your bindings if necessary.
   b. For each resource define an authentication type, which is the security configuration through which database connections are granted. There are two authentication types:

      Component-managed
      The enterprise bean code performs EIS signon for data source or connection factory connections.

      Container-managed
      The product performs EIS signon.

      See the topic, J2EE connector security, for detailed reference on resource authentication.
5. Configure access intent policy settings for your enterprise beans.
   a. Right-click your EJB module in a Project Explorer view and click **Open With > Deployment Descriptor Editor**.
   b. In an EJB Deployment Descriptor editor, select the **Access** tab.
   c. Under **Isolation Level**, click **Add**.
   d. Select the isolation level, enterprise beans, and method elements. For information on isolation levels, press **F1**.
   e. Click **Finish**.
6. Map enterprise beans to database tables.

Results

Files for the updated application are shown in the Project Explorer view.

What to do next

After testing your application, you are ready to deploy your application to an application server.

Creating or changing a resource reference

A resource reference supports application access to a resource (such as a data source, URL, or mail provider) using a logical name rather than the actual name in the runtime environment. This capability eliminates the necessity to alter application code when you change the resource runtime configurations.

Before you begin

This topic guides you through updating the resource references of an enterprise application that you assembled previously. The topic, Assembling applications, details the assembly procedure.

About this task

Resource references are declared in the deployment descriptor by the application provider. At some point in the application deployment process, you must bind the resource reference to the actual name of the resource in the run time environment. When you create a connection factory or data source in the application server, the application server provides a JNDI name that a component can use to access that connection factory or data source. The application server uses an indirect name with the java:comp/env prefix. For example:

- When you create a data source, the default JNDI name is set to jdbc/*data_source_name*.
- When you create a connection factory, its default name is eis/*j2c_connection_factory_name*.

If you override these values by specifying your own, retain the java:comp/env prefix. An indirect JNDI name allows the connection management infrastructure to access to any data from the resource reference that is associated with the application. This allows you to better manage resources based on the settings for authentication, isolation level, sharing scope, and resolution control.

This topic describes how to update the resource references of an enterprise application using an assembly tool. After you define the resource reference, you can perform an indirect JNDI lookup using the java:comp/env context.

Procedure

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules.
3. Import the enterprise application (EAR file) that you want to change into the EJB project.
4. Display the resource references for the type of module:
   - If an enterprise bean uses the resource reference:
     a. Expand the name of the EAR file.
     b. Expand **EJB Modules**.
     c. Expand the EJB module wanted.
     d. Expand the section for the appropriate type of enterprise bean (**Session Beans** or **Entity Beans**).
     e. Expand the enterprise bean.
   - If a servlet uses the resource reference:
     a. Expand the name of the EAR file.
     b. Expand **Web Modules**.
     c. Expand the web module wanted.
   - If an application client uses the resource reference:
     a. Expand the name of the EAR file.

b. Expand **Application Clients**.
c. Expand the application client module wanted.

5. Right-click the module whose resource references you want to change and click **Open With > Deployment Descriptor Editor**.
6. For servlets and application clients, click **Add**. For EJB modules, select the particular bean and click **Add**.
7. Select the resource reference option and click **Next**.
8. Specify the settings for the resource reference, and click **Finish**.
9. Optional: Select the **References** tab and, under **WebSphere Extensions**, select an isolation level. If you choose to forego this step, the isolation level defaults to TRANSACTION_NONE.
10. Optional: Under **WebSphere Bindings**, specify a JNDI name. If you choose to forego this step you can set (or override) the binding when the application is deployed.
11. Close the deployment descriptor editor and save your changes.

Results

Files for the updated module are shown in the Project Explorer view.

What to do next

Verify the contents of the updated enterprise application in the Project Explorer view. Then, deploy your enterprise application.

You can generate EJB deployment code and deploy an EJB module to a target server in one step. In the Project Explorer view, right-click on the EJB project and click **Deploy**. See also the topic, Deploying EJB modules.

Assembling resource adapter (connector) modules

A resource adapter archive (RAR) file contains code that implements a library for connecting with a backend Enterprise Information System (EIS).

Before you begin

This topic assumes that you have created and unit tested a resource adapter RAR file that you want to assemble in an enterprise application and deploy onto an application server.

A Resource Adapter Archive (RAR) file is a Java archive (JAR) file used to package a resource adapter for the Java 2 Connector (J2C) Architecture for the product.

A RAR file can contain the following:
- Enterprise information system (EIS) supplied resource adapter implementation code in the form of JAR files or other runnable components, such as dynamic link lists.
- Utility classes.
- Static documents, such as HTML files, images, and sound files.

The standard file extension of a RAR file is *.rar*.

About this task

In an assembly tool, RAR files are called *connectors* and assembled resource adapters are called *connector modules*.

A *connector* is a Java Platform, Enterprise Edition (Java EE) component that provides access to Enterprise Information Systems (EIS), and must comply with the Java EE Connector Architecture (JCA). An example of an EIS is a transaction manager such as the Customer Information Control System (CICS®).

You might see the terms resource adapter *modules*, resource adapter *connectors* and resource adapter *archive files* used interchangeably.

Use an assembly tool to assemble a *connector* in either of the following ways:
- Import an existing RAR file.
- Create a new connector module.

For information on assembling connectors, refer to the online documentation or the information center for your assembly tool.

Procedure

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on J2EE modules. Ensure that **Java EE** and **EJB** capabilities are enabled.
3. Migrate RAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an assembly tool. To migrate files, import your RAR files to the assembly tool.
4. Create a new connector module.

Results

A connector project is migrated or created. Files for the connector project are shown in the Project Explorer view under **Enterprise Applications** and **Connector Projects**.

What to do next

After creating a connector project, you can edit the connector deployment descriptor if default properties are not sufficient. In the Connector Deployment Descriptor editor, you can view and edit source code.

After assembling the connector project, deploy the module or its application onto a server. After deployment, to ensure that the connector module finds the classes and resources that it needs, check the **Classpath** setting for the RAR on the console Resource adapter settings page.

Planning to use optimized local adapters for z/OS

Use this high-level task when you are planning to implement the optimized local adapters for z/OS in your environment.

Before you begin

For CICS Transaction Server for z/OS to communicate with optimized local adapters, the minimum required version of CICS Transaction Server for z/OS is version 3.1. If you want to use two-phase commit or resource manager local transactions (RMLT) support for outbound calls from WebSphere Application Server to CICS, you must use CICS Transaction Server for z/OS Version 4.1.

About this task

Optimized local adapters and the supporting native API callable services provide a different path for enterprise architecture and application development on the z/OS platform. Optimized local adapters can be used to make inbound calls from an external address space to Enterprise JavaBeans (EJB) applications that are deployed on a local WebSphere Application Server for z/OS server, and make outbound calls from

an application running under WebSphere Application Server for z/OS to a server program that is running in an external address space.

Procedure

1. Review existing business and middleware applications in your environment to determine which processes would benefit from using optimized local adapters. Using optimized local adapters provides applications that are written in native languages like Cobol, PL/I, C/C+, and high-level assembler, and are running under environments such as z/OS batch, Customer Information Control System (CICS) and UNIX System Services (USS), a different way to call Java applications that are implemented as EJB applications on WebSphere Application Server for z/OS.

   You can read about several different types of scenarios where optimized local adapters are used with existing business and middleware applications in the topic, Optimized local adapters for z/OS usage scenarios.
2. Make sure that you are running WebSphere Application Server in 64-bit mode. When you create an application server, it is automatically configured to run in 64-bit mode. If you find that your application server is configured to run in 31-bit mode, you can convert your application server to run in 64-bit mode. To learn how to do this conversion, see the topic, Running servers in 31-bit mode, and the section “Converting a migrated server to run in 64-bit mode”.
3. Make sure that WebSphere Application Server is using a SAF-based user registry if you plan to propagate a SAF user ID from WebSphere Application Server for z/OS to the enterprise information system (EIS).

   Optimized local adapters can be used to connect with CICS and Information Management Systems (IMS™), over OTMA support.
4. Review the optimized local adapters samples. There are several examples that are included when you install WebSphere Application Server for z/OS. See the topic Optimized local adapters Samples on how to locate and use the samples.
5. Decide how to use optimized local adapters. You can use the optimized local adapters to make inbound or outbound calls.
   a. Use the optimized local adapters for inbound support.

      You can use the optimized local adapters support to call inbound to WebSphere Application Server for z/OS EJB applications. The following image shows the flow of an inbound CICS call to a WebSphere Application Server EJB application.

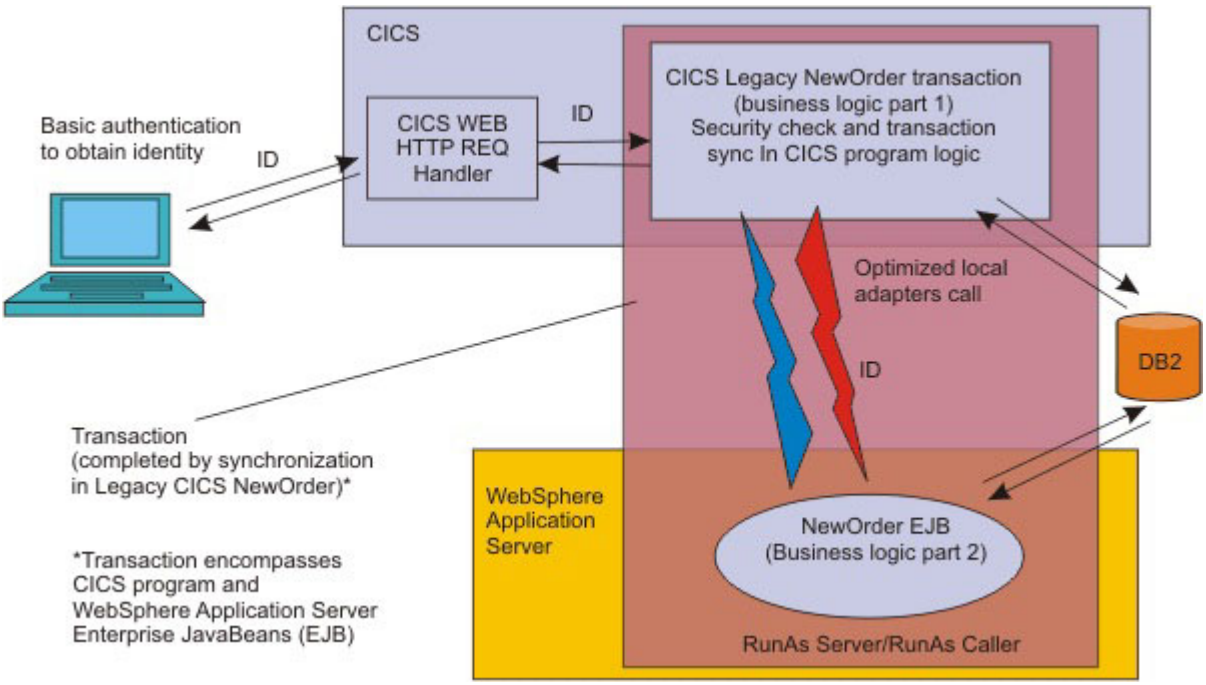


Figure 1. Using CICS

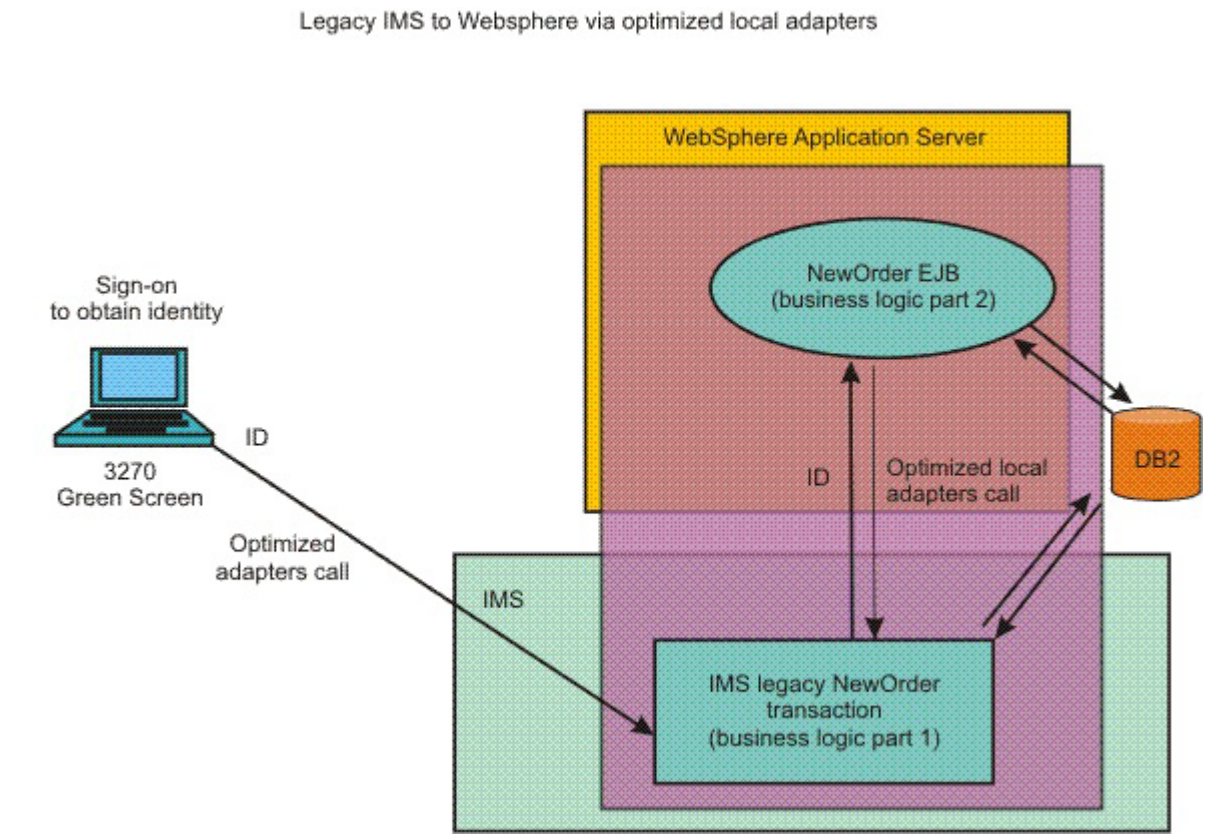
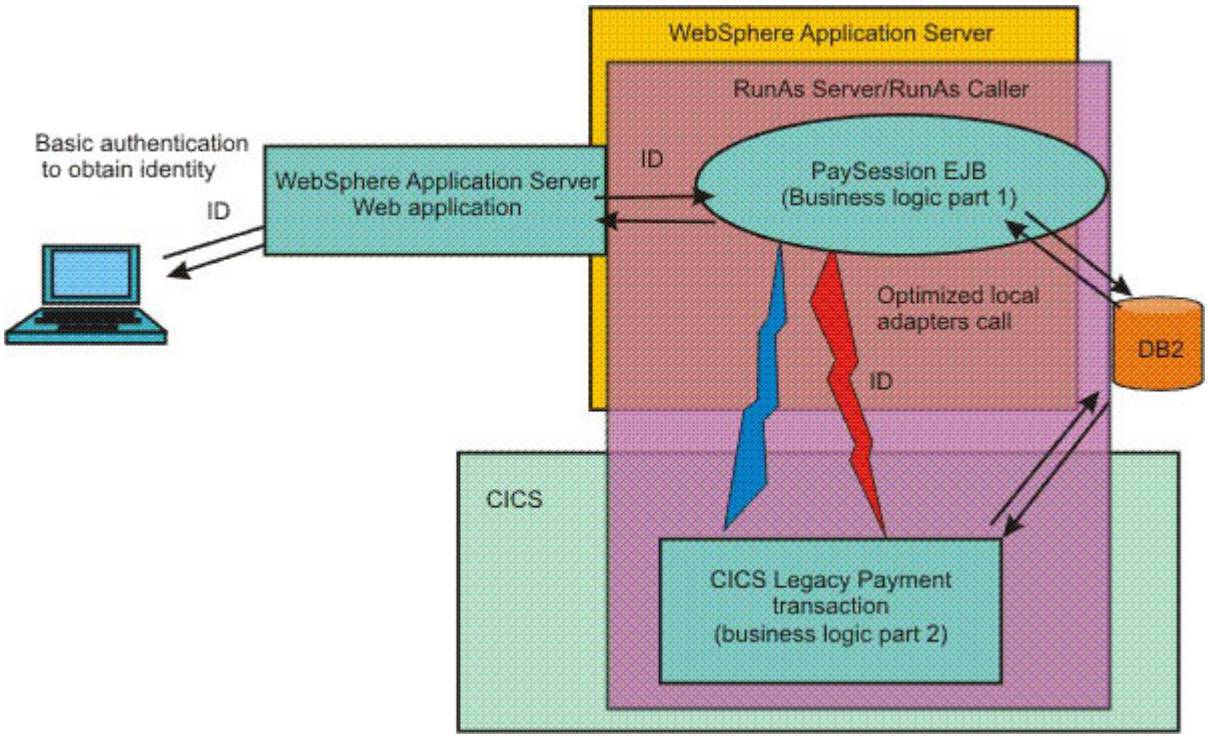


Figure 2. Using IMS

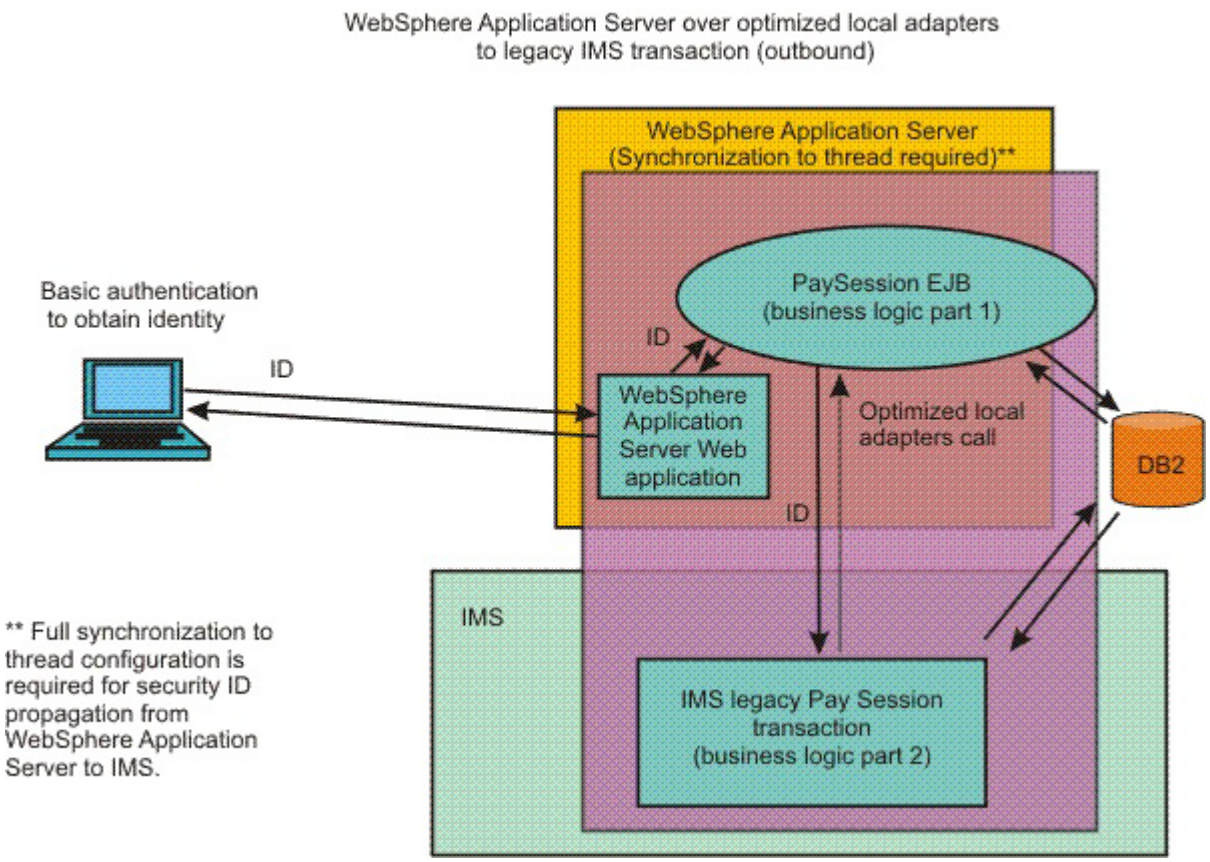
b. Use the optimized local adapters for outbound support.

You can use the optimized local adapters support to call programs in external address spaces from WebSphere Application Server for z/OS applications.

The following illustration shows the flow of an outbound WebSphere Application Server call to a CICS transaction.



The following illustration shows the flow of an outbound WebSphere Application Server call to a IMS transaction.



Optimized local adapters on WebSphere Application Server for z/OS

Optimized local adapters support on WebSphere Application Serverfor z/OS consists of a set of callable services and a Java EE Connector Architecture (JCA) 1.5 resource adapter. The services and adapter work together to provide high performance calling between native language applications on z/OS and business logic in a WebSphere Application Server for z/OS environment.

The optimized local adapters can be used to make inbound calls to applications that are deployed on WebSphere Application Server from an external address space. They are also used to make outbound calls from WebSphere Application Server applications to applications that are running in an external address space on the same z/OS system.

With this support, existing z/OS applications that are written in Cobol, PL/I, C, C++, and assembler can achieve high performance and efficient integration with Java applications that are deployed under WebSphere Application Server on the same z/OS system.

Optimized local adapters also provide close integration of qualities of service (QoS), including support for fast thread-level security propagation and assertion between the API-exploiting external address spaces and WebSphere Application Server for z/OS. Support is provided for using the adapters APIs in the following environments: Customer Information Control System (CICS), Information Management System (IMS), UNIX System Services (USS), and batch processing.

Two paths for the adapters exist: Support for inbound Enterprise JavaBeans (EJB) calls to WebSphere Application Server for z/OS and support for outbound communication with locally running external server programs from WebSphere Application Serverfor z/OS.

A task-related user exit (TRUE) program is provided to support the optimized local adapters under CICS.

Optimized local adapters support is provided for applications running in IMS-dependent region environments using the IMS External Subsystem Attach Facility (ESAF). With this ESAF, WebSphere optimized local adapters are implemented as an IMS subsystem.

Benefits to using optimized local adapters

There are several benefits to using the optimized local adapters, including:

- **Performance improvement**

  Significant performance characteristics can be achieved when using the optimized local adapters APIs to call into applications that are deployed on a WebSphere server from a local batch, USS, IMS, and CICS applications. The ability to pass parameter data using binary techniques provides a large part of the performance improvement. The transport-level support that the adapters provide uses z/OS cross-memory services to optimize the performance of calls to applications that are deployed on a locally accessible WebSphere Application Server for z/OS server.

- **Identity context propagation**

  For inbound requests to WebSphere Application Server using the optimized local adapters APIs, the user ID on the existing z/OS thread is always propagated and asserted in theWebSphere Application Server EJB container. For calls from CICS, this can be extended with a registration option which indicates that the identity of the CICS task level user is propagated and asserted. For calls from applications deployed on WebSphere Application Server, identity can be propagated and asserted under CICS using the optimized local adapters CICS Link server. Selection of this behavior is also controlled with a flag on the registration API.

  When using optimized local adapters over IMS Open Transaction Manager Access (OTMA) support, the identity of the WebSphere Application Server application user on the current thread can be propagated and asserted in the target IMS-dependent region (Fast Path or Message Processing Region).

- **Global transactions**

  Global, two-phase commit transactions are supported with the optimized local adapters for inbound calls from CICS to WebSphere Application Server, and outbound calls from WebSphere Application Serverto CICS.

  Resource manager local transactions (RMLT) are supported with the optimized local adapters for outbound calls from WebSphere Application Server for z/OS to CICS when using the supplied link server.

  Attention: Two-phase commit and RMLT support for outbound calls from WebSphere Application Server to CICS requires the use of CICS Transaction Server for z/OS, Version 4.1 or later.

- **Workload balancing and availability**

  The workload balancing framework in the optimized local adapter support is designed so that the inbound call requests are passed into the target server control region, where the requests are queued using z/OS workload management (WLM) to an eligible servant region for execution.

- **Local binding support**

  Optimized local adapters can provide a high performance local binding for existing applications, middleware, and subsystems on z/OS platforms. These local bindings are used with current programming interfaces when it is determined there is a local WebSphere Application Server available.

- **Providing a gateway or proxy for legacy assets on z/OS systems**

  Built-in optimized local adapters provide the basis for you to begin to use the WebSphere Application Server for z/OS stack as an easily accessible set of capabilities. These capabilities extend the life of application assets that might be difficult to replace. When you use an enterprise bean as a proxy, any Cobol, assembler, or C/C+ application that is deployed on a z/OS system can easily become a Web services client or a Web 2.0 application requestor that reaches a set of Web applications that are within reach of your locally running application server.

  Using the WebSphere Application Server outbound APIs, any Cobol, assembler, or C/C+ application can be presented to WebSphere Application Server as a callable service. A provider Web services application can then be deployed in the local WebSphere server that accepts requests as a gateway for

this backend service. In this scenario, the JCA 1.5 programming model is used to send requests to the application, received responses from it and send back responses to the Web-based caller.

Optimized local adapter Samples

Optimized local adapters Samples

The product provides sample files that support optimized local adapters for z/OS.

The following sample files are located in the WebSphere Application Server product directory, `<Prod_FS_root>/util/zos/OLASamples`:

- `.jclsamp` files
- `olarar.py` and `olararupdate.py` files
- Header files, `bboaapi.h` and `bboaapip.include`
- EAR sample files

The `ola_apis.jar` file is located in the product directory, `/lib`.

Sample descriptions

A directory of the sample names and what they do is available in a directory in the @@README member of the native set of files. The samples include:

- CSDUPDAT - Customer Information Control System (CICS) DFHCSDUP utility job that defines all the resource definitions needed for optimized local adapters under CICS. Group BBOACSD contains CICS definitions for the optimized local adapters; group BBOASAMP contains definitions for the adapter samples.
- DFHPLTOL - Job Control Language (JCL) source for assembling a sample PLT with the optimized local adapters. Enable the Task Related User Exit (TRUE) program, BBOACPLT, and the optimized local adapters BBOC command processor PLT, BBOACPL2.
- OLABATCH - JCL to run one of the samples in batch. Ensure that this runs on the same LPAR that WebSphere Application Server for z/OS is running.
- OLACB01 - JCL source for CICS link to sample Cobol program that uses a commarea. This is a sample target program when using the optimized local adapters CICS Link server. It echoes back the sent message.
- OLACB02 - JCL source for CICS link to sample Cobol program that uses a container. This is a sample target program when using the optimized local adapters CICS Link server. It echoes back the sent message.
- OLACB03 - JCL source for CICS sample Cobol program that demonstrates how to make a CICS task into an optimized local adapters server using the Host Service API.
- OLACB04 - JCL source for CICS sample Cobol program that demonstrates how to make a CICS task into an optimized local adapters server using the Receive Request and Get Data APIs
- OLACB05 - JCL source for CICS sample Cobol program that demonstrates how to use the APIs to Register, Get a Connection, call an Enterprise JavaBeans (EJB) with Send Request, get the response with Get Data, and release the connection with Connection Release and Unregister.
- OLACB06 - JCL source for CICS sample Cobol program that demonstrates how to use the APIs to Register, call an EJB application with Invoke and Unregister.
- OLACC01 - JCL source for a C program that does Register, Invoke, and Unregister. This can be run under batch, UNIX System Services (USS) and CICS.
- OLACC02 - JCL source for a C program that does Host Service, Send Request, Send Response, and Get Data API calls. This program invokes itself, calling in to an enterprise bean that then calls back to this program. This can be run under batch, USS and CICS.
- OLAMAP - JCL source for a CICS BMS screen map definition. This is a 3270 test driving screen for passing requests to and from CICS to WebSphere Application Server.

- OLAUTIL - JCL source for Cobol CICS test application utility. You can test Register, Invoke, Host Service, Send Response APIs from this panel and update the daemon group (cell short name), server and node names, as well as the service name, to run with. This utility can be used for testing calls in both directions. Code from here can be used as samples for using these APIs.
- OLAPL01 - JCL source for a PL/I program that runs as an IMS Fast Path program and is called by the sample WebSphere Application Server application using the optimized local adapter over OTMA support. It in turn does an optimized local adapter Register, Invoke, and Unregister - calling the target EJB in the OLASample2 application.
- OLAPL02 - .JCL source for a PL/I program that runs as an Information Management System (IMS) Message Processing Program (MPP) and is called by the sample WebSphere Application Server application using optimized local adapters over OTMA support. It in turn does an optimized local adapter Register, Invoke, and Unregister - calling the target EJB in the OLASample2 application.
- PSBOLA2 - Sample JCL/source to show how the IMS processor-side bus (PSB) generates the samples OLAPL01 and OLAPL02.
- OTMAINIT - Sample JCL to show how to start the IMS OTMA callable interface SVCs on your system.
- STAGE1 - Sample source for IMS STAGE1 for the samples, OLAPL01 and OLAPL02

Samples installation

1. Allocate a Partitioned data set (PDS) or Partitioned data set extended (PDSE) to hold the JCL source. In the sample JCL, this data set is named BOSS.OLA.SAMPLES.SRC.

   This data set is allocated as RECFM=FB, DSORG=PO, LRECL=80, BLKSIZE=9040, TRKS=40. Copy, for example OGET, the files with file type “jclsamp” from the `<Prod_FS_root>/util/zos/OLASamples` directory to this data set. The header file `bboaapi.h` is also put in this data set as BBOAAPI.
2. There is an enterprise archive (EAR) file that can be installed immediately after you have set up your resource adapter, ola.rar, and defined a connection factory with the Java Naming and Directory Interface (JNDI) name of eis/ola.

   The EAR file is located in the directory, `<Prod_FS_root>/util/zos/OLASamples`.

   Attention: If you are using V8.0 use the `OLASample2.ear` file.
3. Allocate another PDS or PDSE to be used to hold the COBOL COPYBOOK created by the Customer Information Control System (CICS) BMS map build job OLAMAP.

   This data set is allocated as RECFM=FB, DSORG=PO, LRECL=80, BLKSIZE=9040, TRKS=15. In the sample JCL, this data set is named BOSS.OLA.SAMPLES.COPYBOOK.
4. Allocate or select a load module library to contain the optimized local adapter samples load modules.

   It must be allocated as a LIBRARY rather than a PDS.
5. Use the following steps to build and run the z/OS batch samples.

   For all the members you update during these steps, you must change the JCLLIB statement to point at your procedure libraries (for C compiles in this case). Someone familiar with where this information is located on your system should be involved in doing this. Also, the test case source has the daemon group (cell short name), the node short name, and the server short name embedded and you must change these before compiling, in order for these to function on your system.
   a. Edit member OLACC*nn* (where *nn*=test case number) and update the JCL to match your site data set naming conventions for your C compiler and set the SYSLMOD data set where the output load module is placed.
   b. In member OLACC*nn*, update the daemon group name (cell short name), the node name and the server name and submit the job to build the test case load module.
   c. Ensure that the target application server is started with optimized local adapters support enabled. You enable the optimized local adapters support by setting the WAS_DAEMON_ONLY_enable_adapter to `true`. Also, make sure that the `ola.rar` file is installed and a connection factory is created with the JNDI name, eis/ola.
   d. Ensure that the OLA sample EAR file is installed in the target application server.

Attention: If you are using V8.0, use the `OLASample2.ear` file.

e. Use OLABATCH as a template for running the OLACC*nn* batch samples.

Some sample jobs refer to a data set named BOSS.OLA90902.SBBOLOAD. This represents the data set into which you copied the online adapter modules using the **copyZOS.sh** script.

6. The z/OS batch samples can also be run under CICS. Follow the steps in the topic, Enabling optimized local adapters support in CICS, and add the samples load modules library from step 4 to the CICS DFHRPL DD concatenation.
7. Use the following steps to build and run the optimized local adapters CICS sample test utility panel. For all the members that you update during this process, you need to change the JCLLIB statement to point at your procedure libraries (for COBOL compilers and for CICS translation in this case). Someone familiar with where this information is located on your system needs to be involved in doing this process.
   a. Edit member OLAMAP and update the JCL to match your site data set name conventions for HLASM applications.

      You need to change the MAPLIB and DSCTLIB parameters to your own data set names. The MAPLIB should point to the samples load module library that you allocated in step 4. The DSCTLIB should point to the COPYBOOK data set that you allocated in step 3.

      Note: You can change the default register name on this panel from CICSTEST to something else. You can also set the default daemon group name (cell short name), node name and server name. The values you enter here display on the panel when you run the OLAU transaction under CICS.

   b. Submit the OLAMAP job to build the CICS map set module.
   c. Edit member OLAUTIL and review the content.

      This is a sample Cobol application that demonstrates a number of the optimized local adapters APIs from Cobol. It sends and receives the CICS BMS map, OLAMAP, and can send a message to any target enterprise bean in any locally attached application server. It can also be used to demonstrate how to make your CICS task into an optimized local adapters target service using the BBOA1SRV API. After updating the JCL to conform to your local data set name conventions, submit the job and the OLAUTIL load module is saved in the data set pointed to by the PROGLIB symbol.
   d. Ensure that the optimized local adapters load module library, and the optimized local adapters samples load module library, are in the CICS DFHRPL DD concatenation of your CICS cataloged procedure.
   e. Ensure that the sample job CSDUPDAT has run and the steps to install optimized local adapters under CICS are complete.
   f. Ensure that the target application server is started with the optimized local adapters support enabled. You enable optimized local adapters by setting the WAS_DAEMON_ONLY_enable_adapter to `true`. Also, make sure that the `ola.rar` file is installed and a connection factory is created with the JNDI name eis/ola. You can read more about these procedures in the topic, Enabling the server environment to use optimized local adapters.
   g. Ensure that the OLA sample EAR file is installed in the target application server.

      Attention: If you are using V8.0, use the `OLASample2.ear` file.

   h. Start CICS.

      Make sure that optimized local adapters support is enabled, logon to CICS with a user ID that is authorized to run the BBOC and OLAU transactions, and clear the screen. Enter BBOC START_TRUE to start the optimized local adapters CICS task-related user exit (TRUE). A message displays about the exit starting successfully. If you do not get this message, you should get a message indicating the kind of error that occurred. For more detailed messages, refer to the CICS job output and look in file BBOOUT. If you want to use the optimized local adapters Program List Table for Post Initialization (PLTPI) program to start the optimized local adapters TRUE during CICS startup, refer to the following section that describes this process.

i. Clear the screen again and enter OLAU to start the test panel.

   If everything is working properly, the panel displays with the heading, * Optimized Local Adapters WAS z/OS Testing *. The Run parameters are listed on the panel, with Register first with a value of `Y`, Register name with a value of `CICSTEST` and Service name with a value of `ejb/com/ibm/ola/olasample1_echoHome`. The Number of Tests to run field has a value of `00001`.

j. In the Send message data field, enter a message to send to the service in WebSphere Application Server.

k. Enter the WebSphere Application Server server short name, WebSphere Application Server node short name, andWebSphere Application Server cell short name (daemon group name) for the server that you want to call an enterprise bean into.

l. The remaining fields can remain as they are. The service name displayed above is the JNDI home name of a sample target enterprise bean in the OLA sample EAR file.

   Attention: If you are using V8.0, use the `OLASample2.ear` file.

m. Click **PF4** to send the message to the EJB, olasample1_echoHome. The message returns in the Received message data field.

n. You have now demonstrated a call from a CICS Cobol program to a WebSphere Application Server EJB application.

   The panel display has now changed. The Register First? field changed from `Y` to `N`. Rerunning requests with this Register name does not require a register call first, so this changes to the value, `N`. If you get a return code 8 (RC8) and reason code 8 (RSN8) on Register, this means that you are already registered and do not need to register again. After leaving OLAU and coming back in later, that registration is still active, so you do not need to register again with that name and should set that to the value, `N`.

o. To test a call from WebSphere Application Server to CICS, you can use this same panel.

   You must update the service name field to whatever name you want to identify as your target service name and click **PF5**. This puts the screen into an x-wait since OLAUTIL calls the BBOA1SRV API with the service name and registration name that you requested. The panel shows a registration called CICSTEST and service name called myserv. Other parameters that show values on the panel are WAS server short name, WAS node short name, WAS cell short name (daemon group name), Number of Tests to run, and Number of Tests Completed.

p. When OLAU is in the wait for the service name that you requested, start the sample Web application in your browser.

   Use the following URL (updating the IP/port# for your site): http://nn.nn.nn.nn:nnnn/OLA_Sample1_Web/ - change the *nnnn* port number to your non-SSL WebSphere Application Server application port.

q. A web page displays with the following fields listed: Data to send to external address space, Response back from external address space, OLA Register Name, OLA Service Name, CICS Link server-specific data, CICS Link Request Container ID, CICS Link Response Container ID, and CICS Link Transaction ID. Enter the message that you want to send, register name and service name as you did on the OLAU panel and click **Run WAS->External address space test**.

r. You now see the message you entered on the browser display on your CICS 3270 panel in the Received message data field.

s. Type in a response message in the Send message data field and click **PF6** to send the response back to WebSphere Application Server. This should come up on the browser.

t. You have now demonstrated a call from a WebSphere Application Server servlet to a CICS Cobol program.

8. Use the following steps to demonstrate calling from a WebSphere Application Server application to a CICS Cobol program using the optimized local adapters CICS Link server.

   a. Edit member OLACB01 and review the contents.

      This is a sample Cobol application that is the target of an EXEC CICS LINK with a COMMAREA. It writes the passed COMMAREA message data to the default Cobol standard out (CEEMSG) and

echoes back the message. Update the JCL to point to your local data sets and submit it. The load library that this module is saved in must be in the CICS DFHRPL concatenation.

b. Edit member OLACB02 and review the contents.

   This is a sample Cobol application that is the target of an EXEC CICS LINK with a CONTAINER. It reads the CONTAINER contents and writes it back to the same container. Update the JCL to point to your local data sets and submit it. The load library that this module is saved in must be in the CICS DFHRPL concatenation.

c. Ensure that the sample job CSDUPDAT has run and the steps to install optimized local adapters under CICS are complete.

d. Start CICS.

   Logon to CICS with a user ID that is authorized to run the BBOC, BBO# and BBO$ transactions, and clear the screen.

   Enter BBOC START_TRUE to start the optimized local adapters CICS TRUE.

   A message displays about the exit starting successfully. If you do not get this message, you should get a message indicating what kind of error occurred. For more detailed messages, refer to the CICS job output and look in file BBOOUT. If you want to use the optimized local adapters PLTPI program to start the optimized local adapters TRUE during CICS startup, refer to the section in this topic that describes this process.

e. Clear the screen again and enter the following to start an optimized local adapters CICS Link server task: `bboc start_srvr rgn=olaserver svn=<serverName> dgn=<cellName> ndn=<nodeName> mnc=1 mxc=5 sec=n svc=*`

   This results in a BBO$ task starting with the register name OLASERVER, connecting the specified application server. Make sure to specify the server short name, cell short name and node short name for the server.

f. You are now ready to send a request to link to an existing CICS program. Start the test web page (http://nn.nn.nn.nn:nnnn/OLA_Sample1_Web/). Enter OLASERVER as the register name and OLACB01 as the service name.

g. Click **Run WAS** > **External address space test**. You should get a page back with the same message that was sent to CICS returned. If you look in the CICS CEEMSG DD (in the running CICS job), the message data in UTF-8 displayed.

h. You have now demonstrated a call from a servlet in WebSphere Application Server to a CICS Cobol program OLACB01 passing and returning data in a COMMAREA.

i. Display the browser panel again and change the service name to OLACB02 and click the Use Containers check box.

   Important: You must check the Use Containers check box.

j. Click **Run WAS** > **External address space test**. You should get a page back with the input message echoed back.

k. You have now demonstrated a call from a servlet in WebSphere Application Server to CICS Cobol program OLACB02 passing data using a CONTAINER.

l. If you want to track what is happening more closely, you can stop the link server and restart it with tracing set. By setting TRC=2, you can view the trace messages in the CICS job BBOOUT file. To stop the link server type the following: `bboc stop_srvr rgn=olaserver`. To restart the link server with tracing, type the following: `bboc start_srvr rgn=olaServer svn=<serverName> dgn=<cellName> ndn=<nodeName>` mnc=1 mxc=5 sec=*n* svc=* trc=2

9. Use the following steps if you want to set up CICS to automatically start the optimized local adapters TRUE during CICS start up.

   a. You can code BBOC START_TRUE\ in a CICS sequential terminal (TYPE=SDSCI) to start the optimized local adapters TRUE, or

   b. You can create a CICS region Program List Table for Post Initialization (PLTPI) and have it invoked during CICS region startup.

The sample job DFHPLTOL creates a PLTPI with suffix OL. Run this sample, placing the resulting module DFHPLTOL in a load modules library in the DFHRPL concatenation, and add OL to the SIT PLTPI specified for the CICS region, for example, PLTPI=OL.

Refer to the samples file DFHPLTOL for an example of this. When you run the PLTPI, you should see the following messages on your CICS job log if the optimized local adapters TRUE started properly:

```
+BBOA9920I WAS z/OS OLA CICS PLT init start.
+BBOA9921I WAS z/OS OLA CICS TRUE enabled.
+BBOA9925I WAS z/OS OLA CICS PLT init ending.
```

Optimized local adapters tutorials and reference

YouTube video tutorials

Explore WebSphere Application Server optimized local adapters by watching a series of video tutorials on YouTube.

WebSphere Optimized Local Adapters on YouTube

White papers

WebSphere Application Server for z/OS V7 Optimized Local Adapters Planning Guide and Reference

Begin here! If you are thinking about using optimized local adapters, this white paper is a good place to start. This guide can help you determine how to best use optimized local adapters to meet your needs and how to quickly find samples and design information in the product information center.

The WOLA Native APIs ... a COBOL Primer

This white paper includes a series of simple to advanced exercises that illustrate the inbound and outbound WebSphere optimized local adapter native APIs.

Technical brochure

WebSphere Optimized Local Adapters Technical Brochure

Use this two-page reference document for a quick summary of key facts about optimized local adapters.

Wikipedia

Learn about the history of IBM WebSphere optimized local adapters, the technical foundation, components, and more.

WebSphere Optimized Local Adapters on Wikipedia.

Optimized local adapters for z/OS usage scenarios

Optimized local adapters and the supporting native API callable services provide an alternative path for enterprise architecture and application development on the z/OS platform.

Using optimized local adapters provides existing business and middleware applications that are written in native languages like Cobol, PL/I, C/C+, and high-level assembler, and running under environments such as z/OS batch, Customer Information Control System (CICS), Information Management System (IMS), and UNIX System Services (USS), an alternative way to call Java applications that are implemented as Enterprise JavaBeans (EJB) applications on WebSphere Application Server for z/OS.

Optimized local adapters support is also provided for calling from applications that are running on WebSphere Application Server to an external server program running locally or on the same logical partition (LPAR), using the Java EE Connect Architecture (JCA) programming model Version 1.5. The target external server programs might be business or middleware applications that are developed using Cobol, PL/I, C/C+, or high-level assembly languages.

A scenario where the optimized local adapters can provide increased performance is CICS or IMS support for the use of server and client Web services. The targeted backend applications can call business logic that is located elsewhere in a more efficient manner when using the optimized local adapters instead of XML and SOAP messaging technology. Web services is a scenario where you can improve efficiency by using optimized local adapters. The following hypothetical real-world scenarios describe how the optimized local adapters are useful in various business goals.

Financial services company scenario

An IBM z/OS financial services customer that is running business applications under CICS must decide about purchasing a financial processing application, which provides new support for stock trade real-time reporting to the exchanges. The ability to do this style of real-time reporting can result in increasing revenue for the customer.

The application that does real-time reporting is developed as a Java Enterprise Edition (Java EE)-based application and deployed on WebSphere Application Server on a Windows XP platform. The application offers a set of enterprise beans and associated Web services interfaces that can be called for various kinds of interactions.

A test scenario is developed and successfully implemented to call the Java EE application from a CICS Cobol program. Therefore, the customer decides to move forward and do more rigorous testing. Further testing shows that when this mechanism is pressed by more than 50-100 requests per second, it begins to slow to the point where the response times do not meet the customer requirements. The effort is abandoned until a more realistic approach is available for exchanging information in real time between the CICS business application and the new vendor application.

The optimized local adapters can provide this CICS customer with an option to deploy WebSphere Application Server for z/OS and update the CICS application to use the optimized local adapters Invoke or Send Request API. These APIs provide a way to call EJB applications that are deployed on a local WebSphere Application Server for z/OS server, which calls the business logic for the Web service.

Insurance company scenario

An IBM z/OS insurance industry customer that is running a business application under CICS wants to provide customers with the ability to retrieve and update policy information in real time. This information must be gathered in various ways and from several places, including:

- Information directly gathered from DB2
- Information gathered by calling a program in CICS
- Information gathered by starting a Web service to communicate with a remote service provided by another company

The customer chooses to use a Java application for several reasons, but most importantly because most of their programming skills are Java-based. When the new application is tested, the customer experiences long response times when retrieving information. The slow response time is a result of WebSphere Application Server running on a distributed server and the latency involved with communicating remotely with DB2 while calling CICS using Web services and SOAP messages.

To fix the problem, the customer deploys multiple WebSphere Application Server in the same configuration to reduce the number of requests per second on any one of the servers and to spread the requests across separate network paths.

Using optimized local adapters gives the customer an alternative other than deploying multiple servers. The customer could install WebSphere Application Server for z/OS and install the new application on a server on z/OS, closer to the DB2 and CICS environments. For the calls to CICS from WebSphere Application Server, using the optimized local adapters APIs provides a significant boost over the Web services and SOAP solution. Consolidating like this on z/OS platforms reduces the need for more distributed servers that consume floor space, power, and resources to maintain. In this scenario, since the location of the data and applications is the gating factor, increasing the size of the remote server to the most robust one available does not necessarily solve the problem.

Migrating business logic to WebSphere Application Server for z/OS

A customer has years of application logic with Cobol running inside of CICS. They want to migrate some of these applications to WebSphere Application Server to take advantage of Java and Java EE technologies, and use other capabilities in the WebSphere stack.

One of the applications is too large to migrate in one piece, and they would like to gradually move portions of it to WebSphere Application Server. The transactional and security qualities of service provided by CICS must be maintained during the transition, and the performance impact of the transition must be minimal. Using optimized local adapters, portions of the application can be migrated to WebSphere Application Server and wrapped in a stateless session bean. The application logic written in Cobol can be modified to use the optimized local adapter to call the stateless session beans. These calls to WebSphere Application Server run under the same transaction and security contexts used by the Cobol programs running in the CICS region. There is a significant performance gain when compared to making similar calls using a Web service. The customer can continue to relocate portions of the application to WebSphere Application Server until the application is migrated.

Optimized local adapters performance considerations

When using the WebSphere Application Server for z/OS optimized local adapters APIs, there are several areas that need to be considered regarding performance.

The optimized local adapter APIs are designed to provide optimal performance for calling between an external address space and applications on WebSphere Application Server for z/OS, and are expected to establish new kinds of application patterns which supports fine-grained interactions between applications in these environments. The following information describes issues to be aware of regarding optimized local adapters and performance. This content is designed to help you understand the configuration options for using the optimized local adapters in order to achieve the best performance. Benchmark results comparing optimized local adapters to other technologies for synchronous calling between WebSphere Application Server and external address spaces on the same system, like SOAP over HTTP, are not documented here. For this information, read the WebSphere Application Server for z/OS Performance Report.

Selection of Connection Minimum and Maximum Connection values for Register API call

Selecting a value that is too high for the minimum number of connections parameter on the Register API call is not recommended and can degrade performance. This results in a call from the external address space to the WebSphere Application Server control region to establish each connection and add it to the optimized local adapters connection pool during the Register API call. When these connections are established with a server, the connections remain until an Unregister API call is received, at which time WOLA disconnects the connections from the server and removes them from the available connection pool. Setting a minimum value too high results in more memory consumed and adds path length cost to the Register and Unregister APIs. Select a minimum connections value that is best for your need. If the

expectation is that there is a potential for hundreds of simultaneous threads sharing a registration, then it might make sense to pay for the cost of the connections during registration. If the expected number of concurrent threads sharing a registration is lower, then it is recommended that the minimum connections value is set lower.

The maximum connections Register API parameter provides a boundary for the number of connections in the optimized local adapters connection pool for a registration. This is not extendable for the life of the registration. Once the number of concurrent Connection Get requests exceeds this value, the calling thread waits the specified number of seconds set on the Wait Time parameter for Connection Get, Invoke, Receive Request Any, or Host Service APIs for a connection to become available. Once this time expires, a return and reason code are passed back indicating a connection handle cannot be acquired for the request before the wait time expired. There is a maximum size for the connection pool for any single registration which can be set. This value is derived by the cell-wide environment variable, WAS_DAEMON_ONLY_adapter_max_conn. The shipped default value for this variable is 100. The value can be changed using the administrative console. You must restart of the daemon after the setting is changed.

Effect of Connection Minimum and Maximum settings on the optimized local adapters CICS Link server

When starting a Link server task under Customer Information Control System (CICS) (using BBOC START_SRVR), if the minimum connections (MNC) and maximum connections (MXC) parameters are not passed, the MNC register setting defaults to `1` and the MXC defaults to `10`. This means that the number of Link invocation tasks (BBO# tasks) that can be started and run concurrently by the starting Link server (BBO$) task is 10. This translates to the number of concurrent threads from WebSphere Application Server that can run and start CICS target programs. The setting for MXC Link server parameter must reflect the expected duration of the typical target CICS programs that are expected to be started under this instance of the Link server. If the target CICS programs are mostly long-running, then a larger MXC value is appropriate to keep requests flowing efficiently from WebSphere Application Server into CICS. If the target programs are short-lived, then a lower MXC setting is more efficient.

Determining the correct MXC setting should also take into account how many WebSphere Application Server servant regions are communicating with a Link server under a specific registration name. If there is a single servant region and it is running with the threading option set to `ISOLATE` then only a single thread can be sending to CICS in that servant at once, so the MXC value would be set to the number of servants times 1 to ensure that there are no bottlenecks. If running with threading set to `LONGWAIT`, where up to 40 threads can be active per servant, depending on the expected number of requests and types of CICS programs being called, long-running or short-lived, the MXC should be set according to the expected number of concurrent requests across the servants to the Link server running for a specific registration name. It might take some experimentation to determine the optimum MXC setting. Start with a lower number and raise it gradually and settle where the throughput is determined to be best.

Shared 64-bit memory

The optimized local adapters support requires that the WebSphere Application Server for z/OS server runs in 64-bit mode. When the daemon group starts the first time with the WAS_DAEMON_ONLY_enable_adapter value set to `true` or `1`, WebSphere Application Server allocates a shared memory buffer in 64-bit above the bar storage and initializes it. The default size of this area is 32MB. This is where the optimized local adapters shared control structures all reside. It is not where message data is cached. Message and context data is flowed between external address spaces and WebSphere Application Server servant regions using WebSphere Application Server for z/OS local communications inter-address space technology, which stages message data in the server address space. For large messages, this is in 64-bit above the bar storage. Currently, WebSphere Application Server local communication supports a maximum message size of 2GB and in the initial optimized local adapters support, this is the largest supported size for single message.

If a misbehaved application continues to call the Register API and loop without calling Unregister and without terminating (where these are automatically cleaned up), it can overflow the optimized local adapters shared memory buffer for the daemon group. If this occurs, API calls are returned with out of memory reason codes. To diagnose this situation, issue the following command on one of the WebSphere Application Server servers in the effected daemon group:

```
F <server_name>,DISPLAY,ADAPTER,REGISTRATIONS
```

From the display output you should be able to determine what job is consuming and not releasing the registrations and rectify the problem by restarting it.

If, after analysis, the default 32MB is determined to be too small to meet the needs of the daemon group, this value can be changed using the WAS_DAEMON_ONLY_adapter_max_shrmem cell-wide environment variable. Changing this value should only be done after careful consideration. It requires a recreate of the optimized local adapters shared above the bar memory buffer, which can only be done with an IPL of the system.

You can get a rough estimate of the amount of memory needed. Each client registration consumes 392 bytes of shared memory, plus 112 bytes of shared memory for each connection. A registration with a maximum of 100 connections consumes about 12 KB of shared memory. Each client thread, which must wait for a connection to become available, (all connections are in use) consumes an additional 80 bytes. Each service being hosted by the registration consumes an additional 336 bytes.

For example, suppose you have 200 registrations in your daemon group. Each registration contains 200 connections, and will have a maximum of 1000 threads waiting for a connection at any time. The total memory consumed by this configuration is about 20 MB. This leaves enough shared memory to host about 38,000 services concurrently or 190 concurrent services per registration.

```
200 Registrations x 392 bytes=78,400 bytes
200 Registrations x 200 connections x 112 bytes=4,480,000 bytes
200 Registrations x 1000 waiters x 80 bytes= + 16,000,000 bytes
--------------------------------
20,558,400 bytes


33,554,432 bytes – 20,558,400 bytes=12,996,032 bytes remaining
/      336 bytes per service
---------------------------------
38,678 Services
```

When increasing or decreasing the size of the shared memory, remember that above-the-bar shared memory is allocated in 1MB sections. WebSphere Application Server rounds the value you specify upwards to the nearest MB.

Controlling the maximum number of concurrent outbound calls from WebSphere Application Server

There is a daemon-wide default setting that controls the maximum number of concurrent outbound from WebSphere Application Server calls for a single registration that is supported with optimized local adapters. The variable for controlling this is WAS_DAEMON_ONLY_adapter_max_serv. The default value is `100`. This means that there can be no more than 100 different target services running under a single registration (concurrent Host Service, Receive Request Any, or Receive Request Specific API calls). If this value is changed, a restart of the daemon is required.

With the default value of `100`, attempts to set up a thread as a 101st server for a specific registration name using one of the three APIs for this purpose results in a non-zero return and reason code indicating the adapter_max_serv was reached. If an application in WebSphere Application Server looks for this service and it is unavailable immediately, the application waits for a default value of `30` seconds before receiving an exception indicating a timeout occurred waiting for the requested service. In theWebSphere Application

Server servant log this appears as a C9C24C15 minor code. The default 30 seconds for this timeout can be modified by the application using the setConnectionWaitTimeout() method on the Java EE Connector Architecture (JCA) ConnectionSpecImpl.

Optimized local adapters CICS Link server performance considerations

The optimized local adapters support for the CICS Link server can be used to provide a simple means for invoking existing CICS application programs from applications running on WebSphere Application Server for z/OS. When you start the Link server using the BBOC transaction, or the CICS PLTPI program BBOACPL2, the optimized local adapters Link server task (BBO$) starts and receives program link requests from WebSphere Application Server. The Program Link task (BBO#) is then initiated, which in turn issues an EXEC CICS LINK to the target program, receives the response, and sends it back to the WebSphere Application Server caller. Part of this support involves propagating and asserting the WebSphere Application Server application thread-level identity onto the target CICS task. The propagation and assertion of the identity is requested using the SEC=Y parameter on the BBOC START_SRVR command.

Running a Link server with SEC=N and using the identity of the user ID that initiated the Link server yields the better performance, but might not be aligned with the security and auditing requirements of your organization.

If it is determined that the Link server can run with SEC=N, the best performance is achieved by also running with the REU=Y BBOC START_SRVR parameter. REU=Y results in the Link server reusing the program Link invocation tasks (BBO# transactions) between program invocation requests.

Important: If you run the Link server in this configuration, the support in the optimized local adapters JCA for passing a separate LINK transaction ID for individual requests is disabled and a request for this results in a ResourceException thrown back to the application. Also, if you attempt to select REU=Y and SEC=Y, the reuse option is forced to No as the Link server must start a new Program Link task for each request with the identity that was propagated asserted.

Running with the REU=Y option means that the Program Link tasks (BBO#s) remain active, once started, until a BBOC STOP_SRVR or BBOC UNREGISTER is entered for the registration. If you are running with a high MXC value on BBOC START_SRVR and a large number of requests arrive concurrently, the number of BBO# tasks can get high and these do not terminate until the Link server is stopped. This is another issue to consider when determining whether to use REU=Y and what is an appropriate MXC value.

If the goal is to achieve the fastest performance for calling into an application under CICS from WebSphere Application Server, consider coding the Host Service (BBOA1SRV), Receive Request Any (BBOA1RCA), or Receive Request Specific (BBOA1RCS) APIs directly in your application program. With this, there is no built-in support for identity propagation as the Link server provides, but if this is not required and performance is a high enough priority, then direct use of the APIs might be the best option.

JCA considerations

When using the optimized local adapters JCA resource adapter, keep in mind that there is additional overhead with each connection that is obtained from the ConnectionFactory object. If your application must make several calls to an external address space or CICS in the same application method, using the same connection for each interaction performs better than obtaining a different connection for each interaction. In addition, a JCA interaction object can be used repeatedly within the same application method.

When creating the JCA ConnectionFactory for optimized local adapters, it is possible to modify the minimum and maximum size of the JCA connection pool for that ConnectionFactory. This connection pool represents logical connections, which are bound to physical connections (those specified on your BBOA1REG register call) during an interaction. For optimal performance, the size of the JCA connection

pool should be the same size of the physical connection pool set during BBOA1REG. If your JCA connection pool is set too small, your application might have to wait for a JCA connection object, even though there are physical connections available. Your physical connection pool is shared by all servant regions, so if you have multiple servant regions, you might want to decrease the size of the JCA connection pool for each servant region to keep the total number of JCA connections across all servant regions in line with the size of the physical connection pool.

Developing applications that use optimized local adapters

Using the optimized local adapters native APIs to invoke an EJB application from an external address space

Use this task when you want to use the optimized local adapters native APIs to connect an external address space to WebSphere Application Server for z/OS and invoke an Enterprise JavaBeans (EJB) application that is deployed on the application server.

Before you begin

The WebSphere Application Server daemon group must be active on the same z/OS image that the register request originates from.

When running under Customer Information Control System (CICS), the optimized local adapters task related user exit (TRUE) program must be activated before a connection is made between CICS and WebSphere Application Server. To read about how the TRUE program is activated by the transactions, see the topic, Installing the BBOC, BBO$ and BBO# transactions on the client environment, and the topic, WebSphere Application Server transactions BBOC, BBO$, BBO#. For programs running in z/OS batch and UNIX Systems Services (USS), activating the TRUE program is not required. Ensure that the current address space has already registered and bound to the target WebSphere Application Server daemon group with a call to the Register, BBOA1REG API.

About this task

The adapter APIs invoke a stateless session bean is from an external native language program and retrieves the response. This is designed for exploiters wanting more flexibility and where the response area maximum length is not known before-hand.

Procedure

1. The client address space native language application, such as Cobol, PL/I, C/C++, assembler program, calls the BBOA1CNG Connection Get API and passes the register name it used for the register call. A connection handle is returned that must be used for all future API calls.
2. The client application gathers its parameters and designates the target service name as the Java Naming and Directory Interface (JNDI) home interface path name for the enterprise bean that it wants to invoke and calls the BBOA1SRQ Send Request API. This results in a connection to the WebSphere Application Server control region and then to a Workload Manager (WLM)-derived servant region where the passed JNDI home interface executes its create method. The preset method, execute, is located and invoked with the byte array parameters. Control returns immediately back to the caller if the asynchronous parameter is specified and set to `1`. When the asynchronous parameter is set to `0` (zero), the API returns the length of the response in the ResponseLength parameter, as well as the return value.
3. In the WebSphere Application Server servant, the execute method of the target bean invokes business logic. The execute method of a target bean can now invoke the business logic it requires before returning the response data as a serialized byte array back to the native language caller.
4. A 0 (zero) return code and reason code indicates that the Client API Send_Request was queued successfully. With the asynchronous parameter set to `0` (zero), the response length is provided in the

ResponseLength parameter, as well as the return value. With the asynchronous parameter set to 1, the response might not be ready and a call to the BBOA1RCL Receive_RespLen API is required to determine if the response arrived and the length response.

5. For asynchronous Send_Request calls, the client application calls the BBOA1RCL Receive_RespLen API with either asynchronous 0|1 call. The asynchronous 0 (zero) call indicates that the adapter API must block the thread until a response is received. The asynchronous 1 call indicates that the adapter API returns immediately whether the response has arrived or not.
6. A 0 (zero) return code and reason code indicate that the Receive_RespLen client API call successfully completed. With the asynchronous parameter set to 1, a ResponseLength and return value of all 0xFFs indicates there is no response received yet on the passed connection. This provides the client application with more control over the way it sends requests and receives responses. The client can group send requests and send them in sequence over a group of connections and then periodically poll these connections for responses. A connection that is processing a Send_Request with the asynchronous 1 call, cannot be passed another Send_Request for the same connection until the related Receive_RespLen and Get_Data API calls are processed.

   Important: Using these APIs asynchronously with the asynchronous parameter set to 1, grouping send requests, and processing them simultaneously, can only be done on connections that are configured as non-transactional. For example, when using CICS, you want to align a unit of work with an RRS unit of recovery, and a syncpoint under CICS must be propagated to WebSphere Application Server, there can only be a single connection to a specific WebSphere server in the current CICS task. That request receives a warning return code, which indicates that the adapter does not propagate the commit across more than one connection to the same server in the same CICS task.
7. Client applications use the response length that is returned by Receive_RespLen API to ensure it has an area large enough to hold the response data and uses the BBOA1GET Get Data API call to copy the response data into its buffer.
8. Client applications repeat this using the same connection handle until it is ready to release the connection. When the connection is released, the BBOA1CNR Connection_Release is called. Connection handles must be released before they can be retained on a Connection Get API call and used again.

Results

The client invoked a stateless session bean from WebSphere Application Server using the optimized local adapter APIs.

Using the Invoke API to call an enterprise bean from an external address space

Use this task when you want to use the Invoke API to call a stateless session bean from an external address space that is in an application that is deployed on a locally attached WebSphere Application Server for z/OS.

Before you begin

The WebSphere Application Server daemon group must be active on the same z/OS image that the register request originates from. Ensure that the current address space has already registered and is bound to the target WebSphere Application Server cell using the Register API.

About this task

This method is designed for high-level exploiters looking for a simplified path where the response area length is known beforehand.

Procedure

1. Set up the client address space native language application, such as Cobol, PL/I, C/C++, or assembler program, to gather its parameters and designate the target service name as the Java Naming and Directory Interface (JNDI) home interface path name for the stateless session bean that it wants to start.
2. Call the Invoke (BBOA1INV) API. In the target Enterprise JavaBeans (EJB) application, the execute method runs and then starts any business logic that is required before returning the response data as a serialized byte array back to the native language caller.
3. Review the response data. A `0` return code and reason code indicates that the client Invoke API completed successfully and the response data and response data length are saved in the areas designated by the caller.
4. The client application repeats these steps using the same register name call, calling as many stateless session beans as needed, and as often as needed.

   Attention: With this API, the three primitive functions, Send_Request, Receive_RespLen, and Get_Data are all completed using the Invoke API. The calling thread is blocked until the response data is received and copied to the response area.

Results

The client started a stateless session bean from WebSphere Application Server using the optimized local adapter Invoke API.

Calling an enterprise bean from an external address space within a client-initiated transaction

Use this task when you call an Enterprise JavaBeans (EJB) application that is deployed on WebSphere Application Server for z/OS from an external address space within a client-initiated transaction. The Customer Information Control System (CICS) environment is the only environment where transactional support is supported.

Before you begin

The client process must be running on a z/OS operating system, and the client environment must support transactional semantics. The connection between the client and the WebSphere Application Server server is configured to support transactions. Also, the client must have called the Register API with the TRANSACTIONAL flag set to the value of `1`.

About this task

This process starts when you begin a transaction in a client environment. The client calls an EJB application that is running on WebSphere Application Server for z/OS and propagates the new transaction to the application server. The semantics for starting a transaction in the client environment varies based on the client environment. Refer to the CICS documentation for information about the semantics for starting a transaction in a CICS client environment.

Procedure

1. Deploy an EJB application on WebSphere Application Server. Use a transaction attribute, such as required, support or mandatory, on the execute method.
2. Start a transaction on the client application using the transactional semantics. The client application performs transactional work that is required in the client environment.
3. Use the Invoke (BBOA1INV) API or the Send Request (BBOA1SRQ) API to make a remote call with the client program to the EJB application that is deployed on WebSphere Application Server. The transaction context propagates to the WebSphere Application Server server and the EJB application runs under the transaction context.

4. Use the transactional semantics of the client environment to commit or end the transaction independent of the outcome of the WebSphere Application Server server transaction.

Results

The new transaction is propagated to the WebSphere Application Server for z/OS server. The client commits the transaction and a single two-phase commit completes the transaction across the two address spaces.

Calling an enterprise bean from an external address space while ignoring the client transaction context

Use this task when you want to call an Enterprise JavaBeans (EJB) application that is deployed on WebSphere Application Server for z/OS from an external address space while ignoring the client transaction context. The only environment where transactional semantics are supported is Customer Information Control System (CICS).

Before you begin

The client process must be running on a z/OS operating system and the client environment must support transactional semantics. The connection between the client and the WebSphere Application Server is configured to support transactions. Also, the client must have called the Register API with the TRANSACTIONAL flag set to the value of 1.

About this task

The semantics for starting a transaction in the client environment varies based on the client environment. Refer to the CICS documentation for information about the semantics for starting a transaction in a CICS client environment.

Procedure

1. Deploy an EJB application on WebSphere Application Server using a transaction attribute of not supported, never or requires new on the execute method.
2. Start the client program transaction using the transactional semantics and perform transactional work that is required in the client environment.
3. Use the Invoke (BBOA1INV) API or the Send Request (BBOA1SRQ) API to make a remote call to the EJB application that is deployed on WebSphere Application Server for z/OS. The transaction context propagates to the WebSphere Application Server server, but the EJB application creates a new local or global transaction context, depending on the transaction attribute used by the EJB application.
4. The WebSphere Application Server server transaction is committed at the end of the execute method.
5. Use the transactional semantics of the client environment to commit or end the transaction independent of the outcome of the WebSphere Application Server server transaction.

Results

The new transaction is propagated to the WebSphere Application Server for z/OS. The server ignores the transaction context and drives the EJB call inside its own unit of work, which commits independent of the client's unit of work when the EJB call returns.

Using optimized local adapters to connect to an application in an external address space from a WebSphere application

Use this task when you want to use the outbound APIs to connect to an application in an external address space from an application that is deployed on WebSphere Application Server for z/OS.

Before you begin

The daemon group, address space, and external address space must be set up to use the optimized local adapters APIs. The external address space must be registered in the daemon group by calling the BBOA1REG API.

The application that is running in the external address space must have established itself as an optimized local adapters server task using one of the APIs, including BBOA1SRV, BBOA1RCA or BBOA1RCS.

The optimized local adapters resource adapter archive (RAR) file, ola.rar, is deployed and configured using the WebSphere Application Server administrative console or the `olaRar.py` script.

Procedure

1. Locate the application deployment descriptor in the application that makes the external call. This is the application that is deployed on WebSphere Application Server for which you want to make an outbound call from.
2. Create a resource reference that points to the optimized local adapter connection factory. The optimized local adapter connection factory is created when the `ola.rar` file is installed on WebSphere Application Server.
3. Locate the connection factory for the optimized local adapter. You can find the connection factory by looking up the resource reference in the Java Naming Directory Interface (JNDI), for example:

   ```
   Context ctx = new InitialContext();
   ConnectionFactory cf = ctx.lookup("java:comp/env/ola");
   ```
4. Create a ConnectionSpecImpl method call and provide the register name to connect to. You can either use the Register name as an attribute on the managed connection factory or use the ConnectionSpecImpl method, setRegisterName, to provide the registration name for the application that is running in the external address space or subsystem that you want to connect to. This must be the same registration name that was provided by the application in the external address space or subsystem using one of the server optimized local adapters APIs, BBOA1SRV, BBOA1RCA, or BBOA1RCS. For example,

   ```
   ConnectionSpecImpl csi = new ConnectionSpecImpl();
   csi.setRegisterName ("MyRES1");
   ```

   Attention: Setting the register name on the ConnectionSpecImpl object is not necessary if the register name was specified using the RegisterName custom property on the ConnectionFactory object.

   Attention: If you want to use the resource adapter high availability feature, you must ensure that your application does not use the setRegisterName method and instead you must configure the target Register name in the managed connection's connection factory attributes. Refer to the topic, Enabling resource adapter high availability support, for more information on how to configure high availability.
5. Optional: If you are calling an Information Management System (IMS) transaction that does not use the optimized local adapter server APIs, BBOA1SRV, BBOA1RCA, or BBOA1RCS, use this step to set the IMS Open Transaction Manager Access (OTMA) parameters. You can either set the IMS OTMA server name, XCF group ID and transaction level as attributes on the managed connection factory, or use the corresponding ConnectionSpecImpl setter methods, setOTMAServerName, setOTMAGroupID, and setOTMATranLevel, to provide this information for the application that is running in the external address space or subsystem that you want to connect to.

   Attention: When you use optimized local adapters over OTMA, the registration name does not have a counterpart on the IMS transaction side. The registration name can be set, but it is not used for optimized local adapter calls over OTMA. For more information, see the topic Calling existing IMS transactions with optimized local adapters over OTMA.
6. Use the connection factory to create a connection, for example:

```
Connection con = cf.getConnection(csi);
```

Results

Your application that is deployed on WebSphere Application Server is connected to an external address space and ready to call the services that are hosted on the external address space.

Using the outbound APIs with the external address space or subsystem

Use this task to create an interaction between the outbound APIs and the external address space or subsystem.

Before you begin

Establish a connection with an external address space or subsystem as described in the topic, Using the optimized local adapters to call an external address space or subsystem from a WebSphere Application Server on z/OS application.

Procedure

1. Optional: Start a resource manager local transaction (RMLT) if you are using a Customer Information Control System (CICS) server task that supports transactions. To use a CICS server task that supports transactions, the keyword TXN=Y is used when starting the server task. You can start an RMLT to encapsulate several interactions in a single unit of work. Your Java Enterprise Edition (Java EE) application component must not be running on a global transaction and must have a LocalTransaction boundary set to Application in its application deployment descriptor. The following example shows how to start the local transaction using the LocalTransaction object, after the deployment descriptor has been changed:

   ```
   LocalTransaction lt = con.getLocalTransaction();
   lt.begin();
   ```

   For more information about how to change the LocalTransaction setting on an application deployment descriptor, see the topic, Configuring transactional deployment attributes.

   If your Java EE application component is configured to use the LocalTransaction boundary,`ContainerAtBoundary`, the connection is automatically enlisted in an RMLT when the connection is obtained. The work that is done when using that connection is automatically completed when the Java EE application component ends.

   If your Java EE application component is starting in a global transaction, the connector is automatically enlisted in the global transaction when the connection is obtained. The work done when using that connection is automatically completed when the Java EE application component ends. The ConnectionFactory object that is used to obtain your connection must have the RegisterName custom property set to participate in a global transaction.
2. Create an interaction using the connection object, for example:

   ```
   Interaction int = con.createInteraction();
   ```
3. Create an InteractionSpecImpl object and set the service name into the interaction spec. The service name describes the name of the method or function that you want to call on the remote system. The service name was provided when the remote system called the host or one of the Receive Request APIs, or when using the Customer Information Control System (CICS) server task is the program name to start inside CICS, for example:

   ```
   InteractionSpecImpl isi = new InteractionSpecImpl();
   isi.setServiceName("MYSERVICE");
   ```

   Attention: When you use optimized local adapters over OTMA, the service name is not defined and the target IMS transaction is specified in the message stream. The service name can be set, but it is not used for optimized local adapters over OTMA. For more information, see the topic, Calling existing IMS transactions with optimized local adapters over OTMA.

4. Run the interaction using the created interaction spec, for example:
   ```
   int.execute(isi, null);
   ```
5. If your application used an RMLT, commit or back out the work after you are finished with your last interaction. Use the commit or rollback methods on the LocalTransaction object, for example:
   ```
   lt.commit();
   ```

Results

The interaction with the external address space or subsystem is complete.

Attention: When using an RMLT or global transaction, if the CICS application issues a sync point during the RMLT or global transaction, the server task issues a BBOX abend during the sync point. This causes the sync point to fail and an ASPx abend is generated by CICS. The CICS application should wait for WebSphere Application Server to issue the sync point operation by calling the commit or rollback method on the LocalTransaction object, or by letting the Java EE application component finish.

Attention: If you want to define the maximum time that a CICS transaction can run when a global transaction context or RMLT is imported into CICS from WebSphere, modify the OTSTIMEOUT parameter value on the transaction definition that is used to start the CICS link server link task. By default, the transaction name is BBO#. If the CICS transaction run time exceeds the time that is specified by the OTSTIMEOUT parameter, the CICS task that is running the CICS transaction abends, causing the entire global transaction or RMLT to roll back. For information about how to code the OTSTIMEOUT parameter on the transaction definition, see the CICS Transaction Server for z/OS Resource Definition Guide.

If your Java EE application component is participating in an RMLT or global transaction, and the application is communicating with a batch program or a CICS link server started with keyword TXN=N, the optimized local adapter connection is not able to establish a connection to the target program. The target program is not able to provide the transaction capabilities required by the Java EE application. An exception displays with minor code 0xC9C24C3B. This code indicates that the target registration name is found but is not enabled for transactions. Only CICS link servers enabled with keyword TXN=Y can support propagating transactions from WebSphere Application Server for z/OS.

Example

The following method shows how to start an interaction with input and output data as a byte array (byte[]):

```
public byte[] driveInteraction(javax.resource.cci.ConnectionFactory cf,
javax.resource.cci.Connection con,byte[] inputDataBytes),throws javax.resource.ResourceException
 {
 // Create an interaction using the optimized local adapter connection
   javax.resource.cci.Interaction i = con.createInteraction();
 // The InteractionSpec describes the service we want to call
   com.ibm.websphere.ola.InteractionSpecImpl isi = new com.ibm.websphere.ola.InteractionSpecImpl();
   isi.setServiceName("MYSERVICE");
 // The input data is specified using an IndexedRecord.  The first
 // slot in the indexed record contains the input data bytes.
   javax.resource.cci.RecordFactory rf = cf.getRecordFactory();
   javax.resource.cci.IndexedRecord ir = rf.createIndexedRecord(null);
   ir.add(inputDataBytes);

 // The interaction returns another IndexedRecord, whose first
 // slot contains the output data bytes.
   javax.resource.cci.Record or = i.execute(isi, ir);
   byte[] outputDataBytes = null;

   if (or != null)
   {
      outputDataBytes = (byte[])((javax.resource.cci.IndexedRecord)or).get(0);
   }
```

```
// Return the output data to the caller
  return outputDataBytes;
  }
```

The following method shows how to start an interaction using a Rational Application Developer generated copybook Record object:

```
/**
* An example of driving an optimized local adapter interaction, using a Rational
 * Application Developer copybook mapping class as input (inputRecord) and receiving
 * a Rational Application Developer copybook mapping as output.
*/
  public javax.resource.cci.Record driveInteraction(
   javax.resource.cci.Connection con,
   javax.resource.cci.Record inputRecord)
   throws javax.resource.ResourceException
  {
// Create an interaction using the OLA connection
   javax.resource.cci.Interaction i = con.createInteraction();
// The InteractionSpec describes the service we want to call com.ibm.websphere.ola.InteractionSpecImpl isi =
  new com.ibm.websphere.ola.InteractionSpecImpl();
  isi.setServiceName("MYSERVICE");
// The Rational Application Developer generated copybook implements
// javax.resource.cci.Record and can be passed directly to the interaction.
// The interaction returns an IndexedRecord, whose first slot contains
 // the output data bytes.
   javax.resource.cci.Record or = i.execute(isi, inputRecord);
   javax.resource.cci.Record outputRecord = null;
   if (or != null)
  {
// In this example, RADGeneratedOutputType is the name of the Rational Application Developer
 // generated copybook mapping class.
   outputRecord = new RADGeneratedOutputType();
// The output bytes are stored in the output record returned on the
// interaction, which is an IndexedRecord.
  byte[] outputDataBytes =
    (byte[])((javax.resource.cci.IndexedRecord)or).get(0);

// To convert the output bytes to another Rational Application Developer generated copybook,
// call the setBytes(byte[]) method of that class.  The class will
// implement the  com.ibm.etools.marshall.RecordBytes interface.
  ((com.ibm.etools.marshall.RecordBytes)outputRecord)
       .setBytes(outputDataBytes);
  }

// Return the output data to the caller
     return outputRecord;
  }
```

Optimized local adapters for z/OS APIs

WebSphere Application Server for z/OS optimized local adapters are supported by a set of z/OS native language callable services, or application programming interfaces (APIs), and the Java EE Connector Architecture (JCA).

These callable services can be used from the following native programming languages:

- Cobol
- C/C++
- PL/I
- High Level Assembler

Each callable service has a 31-bit and a 64-bit version. The naming convention is BBOA1- for 31-bit API stubs and BBGA1- for 64-bit API callers.

A caller running in AMODE 64 can use the AMODE 31 versions of the APIs, provided the parameter list is below the bar and it passes parameter using 31-bit conventions (each parameter's pointer is a 31 bit address). Callers running in 64-bit must be in either C, C++, or assembler, and execute in 64-bit addressing mode. When using the BBGA1* 64-bit APIs from assembler applications, a format 4, or F4SA, 64-bit save area is required, which is used by the optimized local adapters stubs to save and restore the caller's registers. When calling from C or C++, the sample header in *<Prod_FS_root>*/util/zos/OLASamples/bboaapi.h defines the 64-bit APIs as external OS_NOSTACK, which ensures that they are called with the proper linkage and save area conventions.

When the term *local connection* is used in this API documentation, a reference is being made to a cross-memory link that is created for communication between an external address space on the z/OS system and the WebSphere Application Server on the same z/OS system. The client address space must be running on the same z/OS image. The adapter API manages these local connections in pools that are associated with each uniquely registered caller. The 12-character registration name can be used for one set only of connection pools per address space. There is no limit to the number of unique registrations in a single address space. It is limited only by the amount of available storage.

The API services are listed according to user goal, for example, the first section listed is Register. This section explains the APIs that are used in the task of registering an optimized local adapter so that it can be used in a call.

- Register - BBOA1REG/BBGA1REG
- Unregister - BBOA1URG/BBGA1URG
- Connection Get - BBOA1CNG/BBGA1CNG
- Connection Release - BBOA1CNR/BBGA1CNR
- Send Request - BBOA1SRQ/BBGA1SRQ
- Send Response - BBOA1SRP/BBGA1SRP
- Send Response Exception - BBOA1SRX/BBGA1SRX
- Receive Request Any - BBOA1RCA/BBGA1RCA
- Receive Request Specific - BBOA1RCS/BBGA1RCS
- Receive Response Length - BBOA1RCL/BBGA1RCL
- Get Message Data - BBOA1GET/BBGA1GET
- Invoke - BBOA1INV/BBGA1INV
- Host Service - BBOA1SRV/BBGA1SRV
- JCA Adapter APIs

Register

Using the BBOA1REG (for 31-bit callers) and BBGA1REG (for 64-bit callers) APIs, you can register with a local WebSphere Application Server for z/OS daemon group and server. The BBOA1REG and BBGA1REG APIs request that a group of optimized connections to a local WebSphere Application Server daemon group and server be allocated and registered under the specified "registername."

Table 12. BBOA1REG (31-bit caller) and BBGA1REG (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1REG or BBGA1REG	`BBOA1REG ( daemongroupname , nodename, servername , registername , minconn , maxconn , registerflags, rc, rsn )`
	
	`BBGA1REG ( daemongroupname, nodename , servername , registername , minconn , maxconn , registerflags , rc, rsn )`

Parameters

daemongroupname (input)

An entry variable or entry constant containing the name of the WebSphere Application Server for z/OS daemon group to be joined. The entry variable or constant must be passed by reference. The WebSphere Application Server z/OS daemon must be running and initialized on the local system. This must be a null-terminated string of exactly 8 characters.

nodename (input)

An entry variable or entry constant containing the name of the WebSphere Application Server for z/OS node (short name) to be joined. The entry variable or constant must be passed by reference. The WebSphere Application Server z/OS Server must be running and initialized on the local system. This must be a blank padded string of exactly 8 characters.

servername (input)

An entry variable or entry constant containing the name of the WebSphere Application Server for z/OS Server (short name) to be joined. The entry variable or constant must be passed by reference. The WebSphere Application Server z/OS server must be running and initialized on the local system. This must be a blank padded string of exactly 8 characters.

registername (input)

An entry variable or entry constant containing the name to be used to register a set of local connections. Later calls require this name to identify the pool of connections to use. This must be a blank padded string of exactly 12 characters and cannot be used in the current address space.

minconn (input)

An integer containing the initial minimum number of connections to allocate for this registration. The adapter attempts to reserve this number of connections with the associated server during registration.

Important: At minimum, one connection is allocated to bind with the target server, even if MINCONN is specified as zero (0). Specifying 0 is the same as 1.

maxconn (input)

An integer containing the maximum number of connections to allocate for this registration. The adapter attempts to extend the local connection pool up to this number during a Connection Get request when the minimum number of connections are all in use.

registerflags (input)

A 32 - bit flag word containing registration flags.

- **reg_flag_trans** - bit 30

  Contains a 1 if the transactional recovery is to be supported over connections under this registration and a 0 (zero) if the transactional recovery is not to be supported. When the transactional parameter is enabled, the adapter API attempts to register with the z/OS Resource Recovery Services (RRS) and a global transaction supporting two-phase commit is created between WebSphere Application Server and the API exploiter's environment.
- **reg_flag_W2Cprop** - bit 31 or bit 64

  The reg_flag_W2Cprop controls WebSphere Application Server to CICS outbound transaction security propagation.
- **reg_flag_C2Wprop** - bit 29

Propagates the identity on the calling task from CICS to WebSphere Application Server. For transactions that are inbound from CICS to WebSphere Application Server, registration flag reg_flag_C2Wprop controls how the identity is determined. When this bit is turned *on*, the CICS application task identity is used in the WebSphere Application Server server authorization process. When this bit is turned *off*, the CICS region identity is used.

Attention: You must enable WebSphere Application Server for CICS application level security to be requested. Set the WebSphere environment variable, ola_cicsuser_identity_propagate, to `1` through the administrative console to set this type of security propagation.

- **reg_flag_trcmod** - bit 0

  Set reg_flag_trcmod if you want to modify the optimized local adapter trace settings for the requested registration. If you do not set this flag, no change to the trace setting is made, and the system defaults or predefined trace settings that apply to the registration name or job name are used.

- **reg_flag_trcmore** - bit 1 and **reg_flag_trcsome** - bit 2

  With reg_flag_trcmod set to 1, set either reg_flag_trcmore or reg_flag_trcsome to specify a trace setting if you want detailed tracing or coarse level tracing for the registration. Leaving both of them set to "off" with reg_flag_trcmod set to `1` forces no tracing for the registration no matter what pre-definitions or defaults are set.

`rc` (output)

An integer return code indicating success or failure of this call.

`rsn` (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled.
- The minimum connections requested is validated against the server maximum connections, which is configurable using the WebSphere Application Server administrative console. A minimum connection (minconn) value of `3` means that three connections are reserved during the register call. If more than three connections are requested at a given time, the connection pool can increase to the maximum connection (maxconn) value. When the number of all connections to a server reaches the maximum that is permitted, all subsequent Connection Get API requests for that server are rejected, even if the maxconn value is not yet reached.

  Important: Use caution in setting the minimum connections value. Except in specific instances, a large setting is not advised as resources in the WebSphere Application Server control region are reserved for each connection.
- Multiple calls to the Register API can be made under the same address space, thread, or Customer Information Control System (CICS) task, however, they cannot share the same register name. The register name must be a unique name.
- Setting the transactional parameter to `1` results in all connections for this registration running with transactionality. This means that RRS is attached from the current address space. See the section on Propagating Transactions for more details on the requirements and set up for this. Transactionality is only supported between WebSphere Application Server and CICS. Requests for `transactional(1)` from other environments results in a warning during the register call, the flag is ignored, and processing continues.
- Use the Unregister API call to remove this registration and release the connection pool associated with it.

- When the address space that makes a register call terminates, the registration is automatically terminated and the connections are released.

Return and reason codes:

Table 13. BBOA1REG and BBGA1REG APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
	4	The transactional register flag was set to 1 in an environment where the adapter does not support global transactions.	This setting is ignored and processing continues.
8	-	Error - see reason code	
	8	The registration name token already exists.	You must unregister this name before calling the Register API for it.
	10	The maximum connections parameter exceeds the maximum number of connections permitted for any single registration.	The Register API request is rejected. Ensure that the target server maximum optimized local adapter value is large enough to accommodate this and all other requests. For more information about the WAS_DAEMON_ONLY_adapter_max_conn environment variable see the topic, Optimized local adapters custom properties, and the topic, Enabling the server environment to use optimized local adapters.
	12	The specified minimum connections parameter is larger than the maximum.	Ensure that the minimum connections setting is less than, or equal to, the maximum setting.
	14	Out of shared memory while trying to create registration.	Increase the shared memory allocation for the optimized local adapter or issue unregister calls to reduce resource consumption.
	21	An error occurred while attempting to contact with the local WebSphere Application Server.	If you set the reg_flag_C2Wprop bit (bit 29) to 1, ensure that the WebSphere environment variable, ola_cicsuser_identity_propagate, is set to 1.
	25	The transaction manager cannot be initiated.	Call IBM Support for assistance.
	70	Out of shared memory while trying to create a connection.	Increase the shared memory allocation for the optimized local adapter.
	74	Input register name contains a null.	Blank pad the register name before starting the Register API.
12	4	An error occurred while locating the BBOACALL module.	Ensure that the data set that contains the WebSphere Application Server BBOACALL module is available in the STEPLIB, LNKLST.
	10	Unable to locate the selectedWebSphere Application Server daemon group.	Ensure that the WebSphere Application Server daemon and target server are started, verify that the optimized local adapters support is active and try again.
	14	The user ID is not authorized for the requested WebSphere Application Server.	Ensure that the user ID is authorized to the CBIND SAF class for the requested WebSphere Application Server.
	16	The node name or server name is not found.	Ensure that the node name and server name parameters that are being passed are valid and that the server is active.
	23	An error occurred while naming the token.	Call IBM Support for assistance.
	24	An error occurred while establishing the initial WebSphere Application Server local communication connection.	Refer to the WebSphere Application Server region logs for details on the local communication connect call.
	28	The registration identified with this name is not valid.	The specified register name is already registered, but the RGE is missing. Contact IBM support and report the error. A workaround is to call the Unregister API and attempt to call the Register API again.
	30	The daemon group is not running with WAS_DAEMON_ONLY_enable_adapter property set to 1.	Add the variable:WAS_DAEMON_ONLY_enable_adapter=1 through the WebSphere Application Server administrative console.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	68	An attachment to shared memory failed.	Call IBM Support for assistance.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.

Table 13. BBOA1REG and BBGA1REG APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Unregister

Unregister from the local WebSphere Application Server for z/OS daemon group and server using the BBOA1URG (for 31-bit callers) and BBGA1URG (for 64-bit callers) APIs.

Table 14. BB0A1URG API (31-bit caller) and BBGA1URG (64-bit caller) syntax. This API requests that a group of optimized connections to a local WebSphere Application Server daemon group and server be released using the specified register name.

API	Syntax

BBOA1URG or BBGA1URG	`BBOA1URG (registername, unregflags, rc, rsn)`
	
	`BBGA1URG (registername, unregflags, rc, rsn)`

Parameters:

registername (input)

An entry variable or entry constant containing the name to be used to unregister a set of local connections. This must be exactly 12 characters, blank padded, and the same name used on BBOA1REG.

unregisterflags (input)

A 32 - bit flag word that contains unregistration flags.

- Reserved - bit 0-30
- Force (0|1) - bit 31 or bit 64

  Contains a 1 if the unregister request should be forced. By default, an unregister request is complete if all the connections have been returned to the connection pool. If all the connections are not returned to the pool, a warning is returned to the caller. The unregister process is complete when the last connection has been returned to the pool. A second unregister request can be made with the force bit set to 1, which forces the unregister process to complete and all remaining connection handles for that registration are invalidated.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled .
- If the Unregister API is not called, and the address space that made a register call terminates, an unregister call is done automatically and the connections are released.
- Any connection handles for this registration that are active when the unregister call occurs might continue to be valid until they are returned to the connection pool using the BBOA1CNR API. To force the connection handles to be cleaned up, a second unregister call must be made, specifying the force flag. This invalidates all outstanding connection handles.

Return and reason codes:

Table 15. BBOA1URG and BBGA1URG APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
	66	The unregister call is delayed until all the connections are returned to the pool.	Unregistration completes when the last connection is returned to the free pool.
8	-	Error - see reason code	
	8	Registration token name does not exist.	You must register this name before calling the Unregister API for it.
	64	The *force* option cannot be specified until a normal unregister is issued.	Call the Unregister API without specifying the *force* option.
	76	An attempt to communicate with the server failed because the server is no longer running.	Start the server and try the communication again.
	82	An attempt has already been made to unregister this registration.	Wait for the previous unregister request to complete or reissue this unregister with the *force* option.
12	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Connection Get

This API requests that an available connection from the pool created with the selected registration name is returned.

Table 16. BBOA1CNG (31 bit caller) and BBGA1CNG (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1CNG or BBGA1CNG	`BBOA1CNG ( registername, connectionhandle, waittime, rc, rsn )`
	
	`BBGA1CNG ( registername, connectionhandle, waittime, rc, rsn )`

Parameters:

registername (input)

An entry variable or entry constant that contains the name to be used to locate the connection pool from which to retrieve a connection. This must be a blank padded string of exactly 12 characters.

connectionhandle (output)

A 12 - byte connection handle that must be passed on later requests for actions on this connection.

waittime (input)

An integer containing the number of seconds to wait for the connection to complete before returning a connection unavailable reason code. A value of `0` implies there is no wait time and the API waits indefinitely.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled .
- Ensure that a successful register (BBOA1REG) call was completed in the current address space with a matching name before using the Connection Get API.

Return and reason codes:

Table 17. BBOA1CNG and BBGA1CNG APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	The registration name token does not exist.	Ensure that you have registered this name before you try to call the name with the Connection Get API.

Table 17. BBOA1CNG and BBGA1CNG APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	10	The connection is unavailable. The wait time expired before the connection request could be obtained.	The application behavior varies. Wait and try again, or abend this connection. Another option is to increase the maximum connections setting on the Register API call.
	24	After a successful register call, an error occurred when getting a connection from the pool.	Verify that the server is started. If it is not working, restart the server and try the API request again.
	28	Registration is found, but is inactive.	
12	10	Unable to locate the WebSphere Application Server daemon group or server.	Ensure that the WebSphere Application Server daemon and server are started and the local connections support is active and try again.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Connection Release

The Connection Release API requests that a connection is returned to the pool that it was retrieved from and is made available for another requestor.

Table 18. BBOA1CNR (31-bit caller) and BBGA1CNR (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1CNR or BBGA1CNR	`BBOA1CNR ( connectionhandle, rc, rsn )`
	
	`BBGA1CNR ( connectionhandle, rc, rsn )`

Parameters:

`connectionhandle` (input)

A 12 - byte connection handle indicating the previously obtained connection that is to be released back into the connection pool.

`rc` (output)

An integer return code that indicates the success or failure of this call.

rsn (output)

An integer reason code that describes the reason that the call failed.

Usage notes

- Ensure that the WebSphere Application Server for z/OS daemon group and server that are specified on this call are started and the support for the local adapters is enabled.
- Before using the Connection Release API, ensure that a successful register (BBOA1REG or BBGA1REG API) call is completed in the current address space with a matching name, and a successful Connection Get (BBOA1CNG or BBGA1CNG) call completed to obtain the connection that is now to be released.

Table 19. BBOA1CNR and BBGA1CNR APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Unable to locate the WebSphere Application Server daemon group and server. Any resources that are related to this connection have been cleared.	None
8	-	Error-see reason code	
	36	The connection state is not valid.	The connection handle that is used for this request is in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle that is used for this request is not valid. Refer to the API documentation for information about client connection handles.
12	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Send Request

These APIs send a request into the local WebSphere Application Server for processing.

Table 20. BBOA1SRQ API (31-bit caller) and BBGA1SRQ (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1SRQ or BBGA1SRQ	`BBOA1SRQ ( connectionhandle, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, async(0|1), responsedatalen, rc, rsn )`
	
	`BBGA1SRQ ( connectionhandle, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, async(0|1), responsedatalen, rc, rsn )`

Parameters:

connectionhandle (input)

A 12 - byte connection handle that is to be used for this request.

requesttype (input)

An integer containing the request type that indicates the type of work request to process. Supported type values: 1 = local EJB work requests, 2 = Remote EJB work requests.

requestservicename (input)

An EBCDIC character string up to 256 bytes in length containing the name of the service to invoke. For Type=1, EJB, this is the JNDI Home name for the local target EJB. For Type=2, EJB, this is the JNDI Home name for the remote target EJB.

requestservicenamel (input)

An integer containing the length of the service name to start or `0` (zero) if the service name is null terminated.

requestdata (input)

A 31-bit or 64-bit pointer to the address of the start of the request data to send.

requestdatalen (input)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode containing the length of the data to send.

async(0|1) (input)

An integer value that when set to 1 indicates that the caller wants control returned immediately, even though the response length might not yet be known. For async(0), the current thread is requesting to wait for the response to be returned from the WebSphere Application Server and the response length is returned in the **responsedatalen** output argument.

responsedatalen (output)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode that contains the length of the response. This length can then be used by the caller to acquire storage before calling the Get Data API to copy it in. When async is set to 1, indicating the caller wants control back immediately, this is set to all 0xFFs if the response is not yet received.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that theWebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled.
- Ensure that a successful register (BBOA1REG) call was completed in the current address space with a matching name before using the Send Request API.

Return and reason codes:

Table 21. BBOA1SRQ and BBGA1SRQ APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	14	An out of memory condition occurred while saving the message.	There is not enough memory available for Send Request to process the message. The current address space is out of private storage.
	16	The request service name length is not valid.	Correct the program and try again.
	18	The request length exceeds the system limits.	The message size is larger than the WebSphere Application Server size can support. Verify that the size is valid. If the size is valid, ensure that the WebSphere Application Server size is large enough to accommodate the message size.
	26	Global transaction could not begin.	
	32	The API call request type is not valid.	The request type is not valid. Correct the program and try the call again.
	34	The target service is not found.	Ensure that the application that contains the target enterprise bean is installed and started in the target server for WebSphere Application Server.
	36	The connection state is not valid.	The connection handle that is used for the request is in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle that is used for the request is not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	44	An exception or unexpected condition occurred in the target enterprise bean.	Refer to the WebSphere Application Server logs to review the exception data.
	46	An error occurred in the local communication send request call.	Check the WebSphere Application Server logs to determine the error.
12	10	Unable to locate theWebSphere Application Server daemon group or server.	Ensure that the WebSphere Application Server daemon and server are started and the local connections support is active. Retry to locate theWebSphere Application Server daemon group and server.
	14	The API calling vector using the provided connections handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.

Table 21. BBOA1SRQ and BBGA1SRQ APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Send Response

This API sends a response to a request back to the local WebSphere Application Server.

Table 22. BBOA1SRP API (31-bit caller) and BBGA1SRP (64-bit caller) syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1SRP or BBGA1SRP	`BBOA1SRP ( connectionhandle, responsedata, responsedatalen, rc, rsn )`
	
	`BBGA1SRP ( connectionhandle, responsedata, responsedatalen, rc, rsn )`

Parameters:

`connectionhandle` (input)

A 12 - byte connection handle that is to be used for this response.

`responsedata` (input)

A 31-bit or 64-bit pointer to the address of the start of the response data to send.

`responsedatalen` (input)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64 bit mode containing the length of the data to send.

`rc` (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that theWebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled .
- Ensure that a successful register (BBOA1REG or BBGA1REG API) call was completed in the current address space with a matching name before using the Send Response API.
- Ensure that a successful connection get (BBOA1CNG or BBGA1CNG) call completed and the handle is provided as input on this call.
- Ensure that a successful receive Request Any, Receive Request Specific, or Host API call was issued and returned with request data for the connection handle. The connection must be in a state where the Send Response API is valid.

Return and reason codes:

Table 23. BBOA1SRP and BBGA1SRP APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	14	An out of memory condition occurred while saving the message.	There is not enough memory available for Send Response to process the message. The current address space is out of private storage.
	18	The response length exceeds the system limits.	The message size is larger than the WebSphere Application Server size can support. Verify that the size is valid. If the size is valid, ensure that the WebSphere Application Server size is large enough to accommodate the message size.
	34	The target service is not found.	Ensure that the application containing the target EJB is installed and started in the target WebSphere Application Server.
	36	The connection state is not valid.	The connection handle used for this request is in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle used for this request is not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	46	An error occurred in the local communication send request call.	Check the WebSphere Application Server logs to determine the error.
12	10	Unable to locate to WebSphere Application Server daemon group or server.	Ensure that the WebSphere Application Server daemon and server are started and local connections support is active and try the call again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.

Table 23. BBOA1SRP and BBGA1SRP APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Send Response Exception

This API sends an exception response data back to the JCA caller in the local WebSphere Application Server. The response is a ResourceAdapterException with the specified exception response data returned.

Table 24. BBOA1SRX (31-bit caller) and BBGA1SRX (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1SRX or BBGA1SRX	`BBOA1SRX ( connectionhandle, excresponsedata, exresponsedatalen, rc, rsn )`
	
	`BBGA1SRX ( connectionhandle, excresponsedata, exresponsedatalen, rc, rsn )`

Parameters:

`connectionhandle` (input)

A 12 - byte connection handle that is used for the response.

`excresponsedata` (input)

Specifies a 31-bit or 64-bit pointer to the address of the start of the exception response data to send.

`excresponsedatalen` (input)

Specifies a 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode of the length of the exception response data to send. Exception response data is an EBCDIC string describing the error.

`rc` (output)

An integer return code indicating success or failure of this call.

`rsn` (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server that are specified on this call are started and the support for local adapters is enabled.
- Ensure that a successful register (BBOA1REG or BBGAREG API) call completed in the current address space with a matching name before using the send response call.
- Ensure that a successful connection get (BBOA1CNG or BBGA1CNG API) call completed and the handle for this is provided as input on this call.
- Ensure that a successful Receive Request Any or Receive Request Specific or Host API call was issued and returned with request data for the connection handle. The connection must be in a state where the Send Response and Send Response Exception APIs are valid.

Return and reason codes:

Table 25. BBOA1SRX and BBGA1SRX APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	10	The connection handle is in a released or not valid state.	Ensure that a valid connection handle was passed.
	14	An out of memory condition occurred while saving the message.	There is not enough memory available for Send Response Exception to process the message. The current address space is out of private storage.
	16	One or more parameters are not valid.	Verify that all the parameters are valid and try the call again.
	18	The response length exceeds the system limits.	The message size is larger than the WebSphere Application Server size can support. Verify that the size is valid. If the size is valid, ensure that the WebSphere Application Server size is large enough to accommodate the message size.
	20	One or more parameters are not valid.	Verify that all the parameters are valid and try the call again.
	28	The registration that the connection handle was associated with is no longer active.	Call register with the registration name again before making the connection get and send request calls.
	36	The connection state is not valid.	The connection handle used for this request is in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle used for this request is not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	46	An error occurred in the local communication send request call.	Check the WebSphere Application Server logs to determine the error.
12	10	Unable to locate to WebSphere Application Server daemon group or server.	Ensure that the WebSphere Application Server daemon and server are started and local connections support is active and try the call again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.

Table 25. BBOA1SRX and BBGA1SRX APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Receive Request Any

Receive a request from a local WebSphere Application Server on any connection. Receive a request and related data on any available connection in the pool for the passed register name. Returns request data length as an output parameter. A connection handle is also an output parameter returned to the caller. A Get Data API call with the returned connection handle returns the received message data.

Table 26. BBOA1RCA API (31-bit caller) and BBGA1RCA (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1RCA or BBGA1RCA	`BBOA1RCA ( registername, connectionhandle, requestservicename, requestservicenamel, requestdatalen, waittime, rc, rsn )`
	
	`BBGA1RCA ( registername, connectionhandle, requestservicename, requestservicenamel, requestdatalen, waittime, rc, rsn )`

Parameters:

registername (input)

An entry variable or entry constant that contains the name to be used to locate the connection pool from which to retrieve a connection. This must be a blank padded string of exactly 12 characters.

connectionhandle (output)

A 12 - byte connection handle that is returned and must be passed on later requests for actions on this connection.

requestservicename (input/output)

An EBCDIC character string up to 256 bytes containing the name of the service. This is the name of the target service specified on the InteractionSpec by the WebSphere Application Server application. A value of * indicates a receive request for all service names arriving under the current register name.

requestservicenamel (input/output)

An integer containing the length of the service name to start or 0 if the service name is null terminated.

requestdatalen (output)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode is returned containing the length of the data to receive.

waittime (input)

An integer that contains the number of seconds to wait for the connection to complete before returning a connection unavailable reason code. A value of `0` indicates that there is no wait time and that the API waits indefinitely.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled .
- Ensure that a successful register (BBOA1REG or BBGA1REG API) call was completed in the current address space with a matching name before using this API.

Return and reason codes:

Table 27. BBOA1RCA and BBGA1RCA APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	11	Bad data was sent from WebSphere Application Server	
	16	The request service name length is not valid.	Correct the program and try the call again.
	19	Local communication existing data failure	
	21	Local communication preview data failure	
	38	The client connection handle is not valid.	The client connection handle used for this request was determined to be not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	46	An error occurred in the local communication send request call.	Check the WebSphere Application Server logs to determine the error.
	76	An attempt to communicate with the server failed because the server is no longer running.	Start the server and try the communication again.
12	10	WebSphere Application Server daemon group or server cannot be located.	Ensure that theWebSphere Application Server daemon and server are started and that the local connections support is active and try the call again.

Table 27. BBOA1RCA and BBGA1RCA APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	24	After a successful registration call, an error occurred when getting a connection from the pool.	Verify that the server is started. If the server is not started, restart the server and try the API request again.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	44	Bad CP build for service call	
	46	Bad CP get for service cell	
	48	Bad CP free for service cell	
	60	Unable to get SRVQ lock	
	62	Unable to unlock SRVQ lock	
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Receive Request Specific

Receive a request from a local WebSphere Application Server on a specific connection. Receive a request and related data for the supplied input connection handle. Returns request data length. A Get Data API call with the returned connection handle returns the received message data.

Table 28. BBOA1RCS (31-bit caller) and BBGA1RCS (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1RCS or BBGA1RCS	`BBOA1RCS ( connectionhandle, requestservicename, requestservicenamel, requestdatalen, async(0|1), rc, rsn )`
	
	`BBGA1RCS ( connectionhandle, requestservicename, requestservicenamel, requestdatalen, async(0|1), rc, rsn )`

Parameters:

`connectionhandle` (input)

A 12 - byte connection handle that is to be used for the receive request.

`requestservicename` (input/output)

An EBCDIC character string up to 256 bytes containing the name of the service. This is the name of the target service specified on the InteractionSpec by the WebSphere Application Server application. A value of * indicates that set up as a server for all service names arriving under the current register name.

requestservicenamel (input/output)

An integer containing the length of the service name to start or 0 if the service name is null terminated.

requestdatalen (output)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode is returned that contains the length of the request data received. This length can then be used by the caller to acquire storage before calling the Get Data API to copy it in. When async is set to 1, indicating the caller wants control back immediately, this is set to all 0xFFs if the request data has not yet been received. In this case, the API must be called again to retrieve an inbound request.

async(0|1) (input)

An integer value that when set to 1 indicates the caller wants control returned immediately, even though the request length may not yet be known. When async is set to 0, this call waits for a request from the WebSphere Application Server to be received.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that theWebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled.
- Ensure that a successful register (BBOA1REG API) call was completed in the current address space with a matching name before using this API.

Return and reason codes:

Table 29. BBOA1RCS and BBGA1RCS APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	10	The connection handle is in a released state improper state.	Ensure that a valid connection handle passed.
	11	Bad data was sent from the WebSphere Application Server	
	16	The request service name length is not valid.	Correct the program and try the call again.
	19	Local communication existing data failure	
	21	Local communication preview data failure	

Table 29. BBOA1RCS and BBGA1RCS APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	28	The registration that the connection handle was associated with is no longer active.	Call the registration name again before calling connection get and receive request specific.
	36	The connection state is not valid.	The connection handle used for this request is in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle used for this request is not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	46	An error occurred in the local communication send request call.	Check the WebSphere Application Server logs to determine the error.
	76	An attempt to communicate with the server failed because the server is no longer running.	Start the server and try the communication again.
	78	An internal error occurred that caused the connection to select a request that was not a part of the transaction being processed by this connection.	Return the connection to the connection pool. If the problem persists, contact IBM Support.
	80	The transaction that is active on this connection has timed out and the connection state could not be reset so that another request is processed.	Return the connection to the connection pool. If the problem persists, contact IBM Support.
12	10	The WebSphere Application Server daemon group or server cannot be located.	Ensure that WebSphere Application Server daemon and server are started and local connections support is active and try the call again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	44	Bad CP build for service cell	
	46	Bad CP get for service call	
	48	Bad CP free for service cell	
	60	Unable to get SRVQ lock	
	62	Unable to unlock SRVQ lock	
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Receive Response Length

Receive response length is used to retrieve the length of the response data from a prior send request call.

Table 30. BBOA1RCL API (31-bit caller) and BBGA1RCL (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1RCL or BBGA1RCL	`BBOA1RCL ( connectionhandle, async(0|1), responsedatalen, rc, rsn )`
	
	`BBGA1RCL ( connectionhandle, async(0|1), responsedatalen, rc, rsn )`

Parameters:

connectionhandle (input)

A 12 - byte connection handle that is to be used for this request.

async (input)

An integer value that when set to `1` indicates the caller wants control returned immediately, even though the response length might not be known. When async is set to `0`, this call waits for the response to be returned from the WebSphere Application Server and supply the response length in the **responsedatalen** parameter value.

responsedatalen (output)

A 32-bit unsigned value in 31-bit mode and a 64-bit unsigned value in 64-bit mode containing the length of the data received is returned. This length can be used by the caller to acquire storage before calling the Get Data API to copy it in. If async is `1`, this might be returned as all FFs if the response data is not yet received

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled .
- Ensure that a successful register (BBOA1REG or BBGA1REG API) call was completed in the current address space with a matching name before using this API.
- Ensure that a successful Connection Get (BBOA1CNG or BBGA1CNG API) call completed and the handle is provided as input on this call.
- Ensure that a successful Send Request (BBOA1SRQ or BBGA1SRQ API) call was completed before making this call.

Return and reason codes:

Table 31. BBOA1RCL and BBGA1RCL APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	11	Bad data was sent from the WebSphere Application Server.	
	19	Local communication existing data failure	
	21	Local communication preview data failure	
	36	The connection state is not valid.	The connection handle used for this request was determined to be in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle used for this request was determined to be not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
12	10	The WebSphere Application Server daemon group or server cannot be located.	Ensure that the WebSphere Application Serverdaemon and server are started and local connections support is active and try the call again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Get Message Data

This API is used to copy the received message data. On return from this call, the message is removed from the adapter message cache.

Table 32. BBOA1GET (31-bit caller) and BBGA1GET (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1GET or BBGA1GET	`BBOA1GET ( connectionhandle, msgdata, msgdatalen, rc, rsn, rv )`
	
	`BBGA1GET ( connectionhandle, msgdata, msgdatalen, rc, rsn, rv )`

Parameters:

connectionhandle (input)

A 12 - byte connection handle that is to be used for this request.

msgdata (input)

A 31-bit or 64-bit pointer to the address of the start of the data area to copy into. The storage this points to must be in a key that is writable by the caller.

msgdatalen (input)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode containing the length of the data to be copied.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

rv (output)

A 32 - bit integer return value containing the size of the context buffer for this request.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled.
- Ensure that a successful register (BBOA1REG or BBGA1REG API) call was completed in the current address space with a matching name before using the Send Request API.
- Ensure that a successful connection get (BBOA1CNG or BBGA1CNG API) call completed and the handle is provided as input on this call.
- If the API caller input **msgdatalen** parameter is larger than the actual message response, the return code contains a `0` and the actual message length is provided to the return value.
-

  Important: Once the get message data call returns to the caller, the message data does not persist and the connection is returned to a state where it can be used for another send or receive request call. A subsequent call to get message data or receive response length is returned in error.

Return and reason codes:

Table 33. BBOA1GET and BBGA1GET APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	36	The connection state is not valid.	The connection handle used for this request was determined to be in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle used for this request was determined to be not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	48	An error occurred in the local communication read request.	Check the WebSphere Application Server logs to determine the error.
	50	The connection with the WebSphere Application Server is terminated.	Check the WebSphere Application Server logs to determine the error.
	72	Response length input parameter is not large enough to contain the response message.	Only a portion of the message is returned. The remainder is discarded. Refer to the return value for the size of the message response.
12	10	The WebSphere Application Server daemon group or server cannot be located.	Ensure that the WebSphere Application Server daemon and server are started and the local connections support is active and try the call again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Invoke

This API uses other underlying primitive API functions to call a method in a local WebSphere Application Server. It is designed to be used in situations where the response output area maximum size is known in advance.

Table 34. BBOA1INV (31-bit caller) and BBGA1INV (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1INV or BBGA1INV	`BBOA1INV ( registername, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, responsedata, responsedatalen, waittime, rc, rsn, rv )`
	
	`BBGA1INV ( registername, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, responsedata, responsedatalen, waittime, rc, rsn, rv )`

Parameters:

registername (input)

An entry variable or entry constant containing the name to be used to locate the connection pool to retrieve a connection for this invocation. This must be a blank padded string of exactly 12 characters.

requesttype (input)

An integer containing the request type that indicates the type of work request to process. Supported type values: 1 = local EJB work requests, 2 = Remote EJB work requests.

requestservicename (input)

An EBCDIC character string up to 256 bytes in length containing the name of the service to invoke. For Type=1, EJB, this is the JNDI Home name for the target.

requestservicenamel (input)

An integer containing the length of the service name to start or `0` if the service name is null terminated.

requestdata (input)

A 31-bit or 64-bit pointer to the address of the start of the request data to send.

requestdatalen (input)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode containing the length of the data to send.

responsedata (input)

A 31-bit or 64-bit pointer to the address of the start of the response data area to copy into. The storage this points to must be in a key that is writable by the caller.

responsedatalen (input)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode containing the length of the data to send.

waittime (input)

An integer that contains the number of seconds to wait for the connection to complete before returning a connection unavailable reason code. A value of 0 (zero) indicates that there is no timeout and that this API waits indefinitely.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

rv (output)

A 32 - bit integer return value containing the size of the message data that was received and copied into the caller response area.

Usage notes:

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled .
- Ensure that a successful register (BBOA1REG or BBGA1REG API) call was completed in the current address space with a matching name before using the Send Request API.
- If the API caller input **responsedatalen** parameter is larger than the actual message response, the return code contains a 0 (zero) and the actual message length is provided in the return value.

Return and reason codes:

Table 35. BBOA1INV and BBGA1INV API return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	10	The connection is unavailable. The wait time expired before the connection request is obtained.	The application behavior varies. Wait and retry, or accept this failed Invoke API call. Another option is to increase the maximum connections setting on the Register API call.
	11	Bad data was sent from the WebSphere Application Server.	
	14	An out of memory condition occurred while saving the message.	There is not enough memory available for the invoke to process the message. The current address space is out of private storage.
	16	The request service name length is not valid.	Correct the program and try the call again.
	18	The response length exceeded the system limits.	The message size is larger than the WebSphere Application Server size can support. Verify that the size is valid. If the size is valid, ensure that the WebSphere Application Server size is large enough to accommodate the message size.
	19	Local communication existing data failure.	
	21	Local communication preview data failure.	
	24	After a successful register call, an error occurred when getting a connection from the pool.	Verify that the server is started. If it is not working, restart the server and try the API request again.
	26	Global transaction could not begin.	
	28	Registration is found, but is inactive.	
	32	The request type on the API call is not valid.	The request type parameter is not valid. Correct the program and try the call again.
	34	The target service not found.	Ensure the application containing the target enterprise bean is installed and started in the target WebSphere Application Server.

Table 35. BBOA1INV and BBGA1INV API return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	36	The connection state is not valid.	The connection handle used for this request was determined to be in the wrong state. Refer to the API documentation for the rules on connection states.
	38	The client connection handle is not valid.	The client connection handle used for this request was determined to be not valid. Refer to the API documentation for information about client connection handles.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	44	An exception or unexpected condition occurred in the target enterprise bean.	Refer to the WebSphere Application Server logs to review the exception data.
	46	An error occurred in the local communication send request.	Refer to the WebSphere Application Server logs to determine the error.
	48	An error occurred in the local communication read request.	Check the WebSphere Application Server logs to determine the error.
	50	The connection with theWebSphere Application Server is terminated.	Check the WebSphere Application Server to determine the error.
	72	Response length input parameter is not large enough to contain the response message.	Only a portion of the message is returned. The remainder is discarded. Refer to the return value for the size of the message response.
12	10	The WebSphere Application Server daemon group or server cannot be located.	Ensure that the WebSphere Application Server daemon and server are started and local connections support is active and try the call again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG API) might have invalidated the connection pool.
	24	After a successful register call, an error occurred while getting a connection from the connection pool.	Verify that the server is started. If it is not started, restart the server and try the API request again.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

Host Service

Host Service for a local WebSphere Application Server. This API uses other underlying primitive API functions to set up a native language z/OS program as a server and target for optimized local adapter calls from a localWebSphere Application Server. It is designed to be used in situations where the request area maximum size is known in advance.

Table 36. BBOA1SRV (31-bit caller) and BBGA1SRV API (64-bit caller) APIs syntax. The syntax is explained in the Parameters section.

API	Syntax

BBOA1SRV or BBGA1SRV	`BBOA1SRV ( registername, requestservicename, requestservicenamel, requestdata, requestdatalen, connectionhandle, waittime, rc, rsn, rv )`
	
	`BBGA1SRV ( registername, requestservicename, requestservicenamel, requestdata, requestdatalen, connectionhandle, waittime, rc, rsn, rv )`

Parameters:

registername (input)

An entry variable or entry constant containing the name to be used to locate the connection pool to retrieve a connection from for this call. This must be a blank padded string of exactly 12 characters.

requestservicename (input/output)

An EBCDIC character string up to 256 bytes containing the name of the service. This is the name of the target service specified on the InteractionSpec by the WebSphere Application Server application. A value of * indicates set up as a server for all service names arriving under the current register name.

requestservicenamel (input/output)

An integer containing the length of the service name to start or 0 if the service name is null terminated.

requestdata (input)

A 31-bit or 64-bit pointer to the address of the start of the request data received. The storage this points to must be in a key that is writable by the caller.

requestdatalen (input)

A 32-bit unsigned value in 31-bit mode or a 64-bit unsigned value in 64-bit mode containing the length of the data area to receive the message into.

connectionhandle (output)

A 12-byte connection handle that is returned to the caller and used for sending a response for this request.

waittime (input)

An integer that contains the number of seconds to wait for the connection to complete before returning a connection unavailable reason code. A value of 0 (zero) implies that there is no timeout and that the API waits indefinitely.

rc (output)

An integer return code indicating success or failure of this call.

rsn (output)

An integer reason code describing the reason for a failure on this call.

rv (output)

A 32-bit integer return value containing the size of the message request data that was received and copied into the caller area.

Usage notes

- Ensure that the WebSphere Application Server for z/OS daemon group and server specified on this call are started and the support for local adapters is enabled.
- Ensure that a successful register (BBOA1REG or BBGA1REG API) call was completed in the current address space with a matching name before using this API.
- If the API caller input **requestdatalen** parameter is larger than the actual message response, the return code contains a 0 (zero) and the actual message length is provided in the return value.
- The returned connection handle must be supplied on any response for this Host call using the BBOA1SRP or BBGA1SRP API. A subsequent BBOA1SRV or BBGA1SRVAPI call reuses the same connection handle. If so, the API assumes that there is no response for the earlier request.
-

Important: Once the Host Service call returns to the caller, the message data is not persisted and the connection is returned to a state where it can be used for another API request.

Return and reason codes:

Table 37. BBOA1SRV and BBGA1SRV APIs return and reason codes. The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
0	-	Success	
4	-	Warning - see reason code	
8	-	Error - see reason code	
	8	Register name token already exists.	Ensure that the register name passed is valid.
	10	The connection handle is in a released state.	Ensure that a right connection handle passed.
	12	The connection handle is not in the connection that is in the registration name.	Correct to program and try the call again.
	16	The request service name length is not valid.	Correct the program and try the call again.
	18	The response length exceeds the system limits.	The message size is larger than the WebSphere Application Server size can support. Verify that the size is valid. If the size is valid, ensure that the WebSphere Application Server size is large enough to accommodate the message size.
	40	A local communication error occurred.	Check the WebSphere Application Server server log for any local communication error messages.
	46	An error occurred in the local communication send request call.	Check the WebSphere Application Server logs to determine the error.
	48	An error occurred in the local communication read request.	Check the WebSphere Application Server logs to determine the error.
	50	The connection with the WebSphere Application Server is terminated.	Check theWebSphere Application Server logs to determine the error.
	72	Response length input parameter is not large enough to contain the response message.	Only a portion of the message is returned. The remainder is discarded. Refer to the return value for the size of the message response.

Table 37. BBOA1SRV and BBGA1SRV APIs return and reason codes (continued). The following table also recommends appropriate actions.

Return Code	Reason Code	Description	Action
	76	An attempt to communicate with the server failed because the server is no longer running.	Start the server and try the communication again.
12	10	The WebSphere Application Server daemon group or server cannot be located.	Ensure WebSphere Application Server daemon and server are started and local connections support is active and try again.
	14	The API calling vector using the provided connection handle cannot be located or verified.	The connection handle is not valid or an unregister call (BBOA1URG or BBGA1URG API) might have invalidated the connection pool.
	34	A 64-bit API stub was used in an IMS environment.	The 64-bit APIs are not supported under IMS.
	38	A 64-bit API stub was used in a CICS environment.	The 64 bit APIs are not supported under CICS.
	44	Bad CP build for service cell.	
	46	Bad CP get for service cell.	
	48	Bad CP free for service cell.	
	60	Unable to get SRVQ lock.	
	62	Unable to unlock SRVQ lock.	
	86	WebSphere Application Server for z/OS master BGVT could not be located	WebSphere has not been started yet on the current z/OS system image. Ensure the daemon and a node/server are started before starting any WebSphere Application Server optimized local adapters client processes.
	88	WebSphere Application Server for z/OS master client stub table could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.
	90	WebSphere Application Server for z/OS optimized local adapters master client stub table slot could not be located.	A WebSphere Application Server has been started on the current z/OS system image, but is not running at the level where the optimized local adapters client stub table is supported. You are possibly running with a WebSphere Application Server Version 8.0.0.1 optimized local adapters stub interface that is not compatible with the maintenance level of the application server. To use the V8.0.0.1 optimized local adapters stubs, the application server must also be running at the WebSphere Application Server Version 8.0.0. level. Ensure your application is running with the level of the optimized local adapters stubs that are compatible with WebSphere Application Server on this z/OS system.

JCA adapters APIs

For calls from WebSphere Application Server into a batch program or subsystem, the WebSphere Application Server application uses the standard JCA APIs. The objects that are customized for this adapter are as follows:

- ConnectionSpec
- InteractionSpec
- Record I/O

See the related links for additional information about these standard JCA APIs in the information center.

The ConnectionSpec API is used to indicate which register name to communicate with. The register name is created when the Register API is used. This also identifies which subsystem to connect to. A subsystem can have more than one register name.

The name of the Program Link transaction ID can be passed fromWebSphere Application Server to CICS, and is used to run the Program Link invocation task. This must be 4 characters and defined in the CICS

region. You must set it up with the same attributes and program name as the BBO# transaction (program: BBOACLNK). When this transaction ID is passed using the setLinkTaskTranid method, it overrides the BBO# API and any transaction ID that was specified using BBOC with the LTX=*xxx* parameter.

A setUseCICSContainer method is provided on the ConnectionSpecImpl. When this is set to 1, the name and type of a message request CICS container can be passed to CICS using the methods, setLinkTaskReqContid() and setLinkTaskReqContType(0|1), where 0=CHAR and 1=BIT. When this is passed, the target program has its own input data passed in the named container, which is created with the selected type of BIT or CHAR.

The name and type of a message response CICS container can be passed to CICS using the methods, setLinkTaskRspContid() and setLinkTaskRspContType(bit|char). When this is passed, the target program response is expected to come from the named container with the selected type of BIT or CHAR.

For IMS application transactions, use the setOTMAMaxRecvSize(*nnn*) method on the Connection Specification to set the Maximum Message Receive size. To set the Maximum Segments connection level value use the setOTMAMaxSegments(*nnn*) method on the Connection Specification. For more information about using optimized local adapters with IMS over OTMA, see the topic Calling existing IMS transactions with optimized local adapters over OTMA.

InteractionSpec

The InteractionSpec API is used to specify which service is to be started by the target. This is dependent on what subsystem or batch process is started. The process can choose to receive requests for a specific service, or for any service. For CICS, the service name is the name of the program to start in CICS.

Record I/O

Applications can pass data to and receive data from the adapter using the JCA record interface. This adapter API proposes to use the indexed record interface to allow the caller to pass one or more data structures, including Cobol copybooks and C structures, to the external address space. Each index in the list is taken as a separate copybook and passed to the target in that order.

When used to accept return parameters from a function call, the IndexedRecordImpl can be predefined to accept a particular number of parameters, and the class names generated by the assembly tools for the individual copybooks and structures can be provided. The IndexedRecordImpl inflates the serialized copybooks and structures and sets them into the IndexedRecordImpl so that they can be retrieved and used without dealing with the serialized byte[] representations.

To give a general idea of how this API works, only selected methods are shown. All of the methods on the interface are implemented.

Optimized local adapters client-side code relocated to common storage

With WebSphere Application Server Version 8.0.0.1, the amount of optimized local adapters code that needs to be loaded in the external client address space on behalf of the optimized local adapters APIs is reduced significantly.

Prior to WebSphere Application Server Version 8.0.0.1, the optimized local adapters OLAMODS dataset, which contains the optimized local adapters API calling vector modules, had to be present in a job's STEPLIB DD or in the z/OS system link list concatenation. With optimized local adapters in WebSphere Application Server Version 8.0.0.1, this is no longer required. The optimized local adapters BBOA1* and BBGA1* API calling stubs have been enhanced to locate the optimized local adapters calling vector in z/OS common storage, which is located off the WebSphere Application Server global BGVT. Once the

application binds in the optimized local adapters API stubs, it no longer requires the BBOACALL optimized local adapters transfer vector module in the STEPLIB.

When the first WebSphere Application Server Version 8.0.0.1 cell, node or server is started, and the optimized local adapters module vector is loaded into common storage, applications that use the V8.0.0.1 calling stubs can exploit this new capability. WebSphere Application Server z/OS initialization code is enhanced to load a new copy of the common storage-based optimized local adapters module vector only when it detects a level of the vector higher than the one that was already active.

Use of the optimized local adapters API stubs (BBOA1* modules) shipped prior to V8.0.0.1 is still supported on V8.0.0.1. However, use of the V8.0.0.1 stubs on WebSphere Application Server versions prior to V8.0.0.1 is not supported. When you begin using the V8.0.0.1 optimized local adapters calling stubs, ensure that they are used to interact with WebSphere Application Server Version 8.0.0.1 and newer servers only.

The new optimized local adapters lightweight stubs are supported in the batch, Customer Information Control System (CICS), IMS, and USS environments. For CICS, the requirement for placing the OLAMODS dataset in the CICS DFHRPL DD concatenation remains. For IMS, the requirement for placing the OLAMODS dataset in the DFSESL DD concatenation also remains.

Chapter 8. Developing Dynamic caching

This page provides a starting point for finding information about the dynamic cache service, which improves performance by caching the output of servlets, commands, web services, and JavaServer Pages (JSP) files.

Dynamic caching features include replication of cache entries, cache disk offload, Edge-Side Include caching, web services, and external caching. Use external caching to control caches outside of the application server.

Configuring cacheable objects with the cachespec.xml file

Use this task to define cacheable objects inside the `cachespec.xml`, found inside the web module `WEB-INF` or enterprise bean `META-INF` directory.

Before you begin

Enable the dynamic cache. Refer to the Using the dynamic cache service article for more information.

About this task

You can save a global `cachespec.xml` in the application server properties directory, but the recommended method is to place the cache configuration file with the deployment module. The root element of the `cachespec.xml` file is <cache>, which contains <cache-entry> elements.

gotcha: In situations where there is a global `cachespec.xml` file in the application server properties directory, and a `cachespec.xml` file in an application, the entries in the two cachespec.xml files are merged. If there are conflicting entries in the two files, the entries in the in the `cachespec.xml` file that is in the application override the entries in the global `cachespec.xml` file for that application.

The <cache-entry> element can be nested within the <cache> element or a <cache-instance> element. The <cache-entry> elements that are nested within the <cache> element are cached in the default cache instance. Any <cache-entry> elements that are in the <cache-instance> element are cached in the instance that is specified in the **name** attribute on the <cache-instance> element.

Within a <cache-entry> element are parameters that allow you to complete the following tasks to enable the dynamic cache with the `cachespec.xml` file:

Procedure

1. Develop a `cachespec.xml` file.
   a. Create a caching configuration file.

      In the *<app_server_root>*`/properties` directory, locate the `cachespec.sample.xml` file.
   b. Copy the `cachespec.sample.xml` file to `cachespec.xml` in web module `WEB-INF` or enterprise bean `META-INF` directory.
2. Define the cache-entry elements necessary to identify the cacheable objects. See the cachespec.xml file topic for a list of elements.
3. Develop cache ID rules.

   To cache an object, WebSphere Application Server must know how to generate unique IDs for different invocations of that object. The *<cache-id>* element performs that task. Each cache entry can have multiple cache-ID rules that run in order until either a rule returns cache-ID that is not empty or no more rules remain to run. If no cache-ID generation rules produce a valid cache ID, then the object is not cached. Develop the cache IDs in one of two ways:

- Use the *<component>* element defined in the cache policy of a cache entry (recommended). Refer to the cachespec_xml file topic for more information about the *<component>* element.
- Write custom Java code to build the ID from input variables and system state. To configure the cache entry to use the ID generator, specify your IdGenerator in the XML file by using the *<idgenerator>* tag, for example:

```
<cache-entry>
     <class>servlet</class>
     <name>/servlet/CommandProcessor</name>
  <cache-id>
      <idgenerator>com.mycompany.SampleIdGeneratorImpl</idgenerator>
      <timeout>60</timeout>
  </cache-id>
</cache-entry>
```

4. Specify dependency ID rules. Use dependency ID elements to specify additional cache group identifiers that associate multiple cache entries to the same group identifier.

   The dependency ID is generated by concatenating the dependency ID base string with the values returned by its component elements. If a required component returns a null value, then the entire dependency ID does not generate and is not used. You can validate the dependency IDs explicitly through the dynamic cache API, or use another cache-entry *<invalidation>* element. Multiple dependency ID rules can exist per cache entry. All dependency ID rules run separately. See cachespec.xml file topic for a list of *<component>* elements.
5. Invalidate other cache entries as a side effect of this object start, if relevant. You can define invalidation rules in exactly the same manner as dependency IDs. However, the IDs that are generated by invalidation rules are used to invalidate cache entries that have those same dependency IDs.

   The invalidation ID is generated by concatenating the invalidation ID base string with the values returned by its component element. If a required component returns a null value, then the entire invalidation ID is not generated and no invalidation occurs. Multiple invalidation rules can exist per cache-entry. All invalidation rules run separately.
6. Ensure your cache policy is working correctly. You can modify the policies within the `cachespec.xml` file while your application is running. The dynamic cache reloads the updated file automatically. If you are caching static content and you are adding the cache policy to an application for the first time, you must restart the application. You do not need to restart the application server to activate the new cache policy. Refer to the Verifying the cacheable page topic for more information.

What to do next

Typically you declare several *<cache-entry>* elements inside a `cachespec.xml` file.

When new versions of the `cachespec.xml` are detected, the old policies are replaced. Objects that cached through the old policy file are not automatically invalidated from the cache; they are either reused with the new policy or eliminated from the cache through its replacement algorithm.

For each of the three IDs (cache, dependency, invalidation) generated by cache entries, a `<cache-entry>` can contain multiple elements. The dynamic cache runs the *<cache-id>* rules in order, and the first one that successfully generates an ID is used to cache that output. If the object is to be cached, each one of the *<dependency-id>* elements is run to build a set of dependency IDs for that cache entry. Finally, each of the *<invalidation>* elements are run, building a list of IDs that the dynamic cache invalidates, whether or not this object is cached.

Verifying the cacheable page

Use this task to verify that the dynamic cache service has its cache policies configured correctly and is serving cached content.

Before you begin

The dynamic cache service should be enabled. You should have a cache policy developed for your application. Refer to the Configuring cacheable objects with the cachespec.xml file article for more information. You must have servlet caching enabled in the web container. Refer to the Configuring servlet caching article for more information.

About this task

You can verify the cacheable page by invoking the snoop servlet in the default application. If the dynamic cache is working correctly, refreshing the servlet repeatedly results in viewing cached content.

Procedure

1. View the Snoop servlet in the default application by accessing the URI: `/snoop` The Snoop servlet is a part of the default application. Refer to the Default application article for more information.
2. Invoke and reload the URI several times using a different web browser or using different parameters. This action returns the same output for the snoop servlet. The snoop servlet is now operating incorrectly, because it displays the request information from its first invocation rather than from the current request.
3. Inspect the entry in the cache with the dynamic cache monitor. Refer to the Displaying cache information article for more information.

---

cachespec.xml file

The cache parses the `cachespec.xml` file when the server starts, and extracts a set of configuration parameters from each cache-entry element. Every time a new servlet or other cacheable object initializes, the cache attempts to match each of the cache-entry elements to find the configuration information for that object.

The cache-entry elements can be inside the root cache element or inside a cache-instance element. Cache entries that are in the root element are cached with the default cache instance. Cache entries that are in the <cache-instance> element are cached in that particular cache instance. Different cacheable objects have different class elements. You can define the specific object that a cache policy refers to using the name element.

Location

Place the `cachespec.xml` file with the deployment module. Use an assembly tool to define the cacheable objects. See topics about assembling applications. You can also place a global `cachespec.xml` file in the application server properties directory. In situations where there is a global `cachespec.xml` file in the application server properties directory, and a `cachespec.xml` file in an application, the entries in the two cachespec.xml files are merged. If there are conflicting entries in the two files, the entries in the in the `cachespec.xml` file that is in the application override the entries in the global `cachespec.xml` file for that application.

The `cachespec.dtd` file is available in the application server properties directory. The `cachespec.dtd` file defines the legal structure and the elements that can be in your `cachespec.xml` file.

Usage notes

Cachespec.xml elements

The root element of the `cachespec.xml` file is cache and contains cache-instance and cache-entry elements. The cache-entry elements can also be placed inside of cache-instance elements to make that cache entry part of a cache instance that is different from the default.

cache-instance

```
<cache-instance name="cache_instance_name"></cache-instance>
```

The name attribute is the Java Naming and Directory Interface (JNDI) name of the cache instance that is set in the administrative console.

Each cache-instance element must contain at least one cache-entry element. A cache entry that is matched within a cache-instance element is cached in the servlet cache instance that is specified by the name attribute. If identical cache-entry elements exist across cache-instance elements, the first cache-entry element that is matched is used.

cache-entry

Each cache entry must specify certain basic information that the dynamic cache uses to process that entry. This section explains the function of each cache entry element of the `cachespec.xml` file including:

- class
- name
- sharing-policy
- skip-cache
- property
- cache-id

With the current version of WebSphere Application Server, you can define multiple cache policies for a single servlet. For example, if you define multiple mappings for a servlet in the `web.xml` file, you can create a cache entry for each one of the mappings.

class

```
<class>command | servlet | webservice | JAXRPCClient | static | portlet </class>
```

This element is required and specifies how the application server interprets the remaining cache policy definition. The value `servlet` refers to servlets and JavaServer Pages (JSP) files that are deployed in the WebSphere Application Server servlet engine. The `webservice` class extends the servlet with special component types for web services requests. The `JAXRPCClient` is used to define a cache entry for the web services client cache. The value, `command`, refers to classes using the WebSphere Application Server command programming model. The value, `static`, refers to files that contain static content. The following examples illustrate the class element:

```
<class>command</class>
<class>servlet</class>
<class>webservice</class>
<class>JAXRPCClient</class>
<class>static</class>
<class>portlet</class>
```

name

```
<name>name</name>
```

Use the following guidelines for the name element to specify a cacheable object:

- For commands, this required element must include the package name, if any, and class name, including a trailing `.\class`, of the configured object.

  Important: If you specify command caching in the cachespec.xml file, and servlet caching is not enabled for the application server on which the application runs, you will get an error. The application server still attempts to obtain a cache instance; therefore, enabling servlet caching will eliminate the error.
- For servlets and JSP files, if the `cachespec.xml` file is in the WebSphere Application Server properties directory, this required element must include the full URI of the JSP file or servlet to cache. For servlets and JSP files, if the `cachespec.xml` file is in the web application, this required element can be relative to the specific web application context root.
- For web services, include the Universal Resource Identifier (URI) of the Simple Object Access Protocol (SOAP) router that is associated with the web service that you want to cache.

- For web services client cache, the name is the target end point of the cacheable web service or the URI of the SOAP router that is associated with the cacheable web service. You can use the SOAP address location in the Web Services Description Language (WSDL) file to define the name for the web services client cache.
- For static files, if the `cachespec.xml` file is in the WebSphere Application Server properties directory, this required element must include the full URI of the file to cache. If the `cachespecm.xml` file is in the web application, this required element can be relative to the specific web application context root. For a web application with a context root, the cache policy for files using the static class must be specified in the web application, and not in the properties directory.
- For portlets, if the `cachespec.xml` file is in the WebSphere Application Server properties directory, this required element must include the full context path and name of the portlet to cache. If the `cachespec.xml` file is in the web application, this required element is the portlet name that is relative to the specific web application context root.

Tip: The preferred location of the `cachespec.xml` file is in the web application, not the properties directory.

You can specify multiple name elements within a cache-entry if you have different mappings that refer to the same servlet.

The following examples illustrate the name element:

```
<name>com.mycompany.MyCommand.class</name>
<name>default_host:/servlet/snoop</name>
<name>com.mycompany.beans.MyJavaBean</name>
<name>mywebapp/myjsp.jsp</name>
<name>/soap/servlet/soaprouter</name>
<name>http://remotecompany.com:9080/service/getquote</name>
<name>mywebapp/myLogo.gif</name>
```

sharing-policy

```
<sharing-policy> not-shared | shared-push | shared-pull | shared-push-pull</sharing-policy>
```

When working within a cluster with a distributed cache, these values determine the sharing characteristics of entries that are created from this object. If this element is not present, a not-shared value is assumed. On the z/OS platform, you can enable replication between servants in a base application server by using the DynacacheEnableUnmanagedServerReplication and DynacacheUnmanagedServerReplicationType Java virtual machine (JVM) custom properties. When enabling a replication, the default value is `not-shared` . This property does not affect distribution to Edge Side Include processors through the Edge fragment caching property.

Refer to the Configuring cache replication article for more information.

Table 38. Sharing-policy values. Values and description for the sharing-policy.

Value	Description
not-shared	Cache entries for this object are not shared among different application servers. These entries can contain non-serializable data. For example, a cached servlet can place non-serializable objects into the request attributes, if the <class> type supports it. **Note:** The application server will always replicate invalidation entries. They are not affected by the sharing policy.
shared-push	Cache entries for this object are automatically distributed to the dynamic caches in other application servers or cooperating Java virtual machines (JVMs). Each cache has a copy of the entry at the time it is created. These entries cannot store non-serializable data.
shared-pull	Cache entries for this object are shared between application servers on demand. If an application server gets a cache miss for this object, it queries the cooperating application servers to see if they have the object. If no application server has a cached copy of the object, the original application server runs the request and generates the object. These entries cannot store non-serializable data. This mode of sharing is not recommended.
shared-push-pull	Cache entries for this object are shared between application servers on demand. When an application server generates a cache entry, it broadcasts the cache ID of the created entry to all cooperating application servers. Each server then knows whether an entry exists for any given cache ID. On a given request for that entry, the application server knows whether to generate the entry or pull it from somewhere else. These entries cannot store non-serializable data.

The following example shows a sharing policy:

```
<sharing-policy>not-shared</sharing-policy>
```

skip-cache

Takes the name of a request attribute, which if present in the request context, dictates that the response cannot be retrieved from the cache instance that is specified. This property is useful for previewing content in production systems and verifying that the application is working and performing as expected.

```
<cache>
 <skip-cache-attribute>att1</skip-cache-attribute> <!—Applies only to the base cache- ->
...
<cache-instance name="instance1">
<skip-cache-attribute>att2</skip-cache-attribute> <!-Applies only to this instance- ->
...
</cache-instance>
</cache>
```

property

```
<property name="key">value</property>
```

where *key* is the name of the property for this cache entry element, and *value* is the corresponding value.

You can set optional properties on a cacheable object, such as a description of the configured servlet. The class determines valid properties of the cache entry. At this time, the following properties are defined:

Table 39. Property values. Property values and valid classes.

Property	Valid classes	Value
ApplicationName	All	Overrides the JavaEEName application ID so that multiple applications can share a common cache ID namespace.
EdgeCacheable	Servlet	True or false. The default is `false`. If the property is `true`, then the given servlet or JSP file is externally requested from an Edge Side Include processor. Whether or not the servlet or JSP file is cacheable depends on the rest of the cache specification. The permissible components for general edgecacheable cache entries are PARAMETER, HEADER, COOKIE, and PATH_INFOSERVLET_PATH
ExternalCache	Servlet and portlet	Specifies the external cache name. The external cache name needs to match the external cache group name.

consume-subfragments	Servlet, web service, or portlet	True or false. The default is `false`. When a servlet is cached, only the content of that servlet is stored, and includes placeholders for any other fragments to which it includes or forwards. Consume-subfragments (CSF) tells the cache not to stop saving content when it includes a child servlet. The parent entry, the one marked CSF, includes all the content from all fragments in its cache entry, resulting in one big cache entry that has no includes or forwards, but the content from the whole tree of entries. Consume-subfragments can save a significant amount of application server processing, but is typically only useful when the external HTTP request contains all the information needed to determine the entire tree of included fragments.
		
		Use the <exclude> element to tell the cache to stop consuming for the excluded fragment and instead, create a placeholder for the include or forward. For example, exclude `A.jsp` from the consume-subfragment, as follows:
		`<property name=`
		` "consume-sbufragments">true`
		`<exclude>/A.jsp<exclude>`
		`</property>`

do-not-consume	Servlet, web service, or portlet	True or false. The default is `false`. When a fragment parent has the consume-subfragment property set to true the child fragment content is saved in the cache entry of the parent. Do-not-consume (DNC) tells the cache to stop saving the content for this fragment in the parent cache-entry and create a placeholder instead for the include or forward.
alternate_url	Servlet	Specifies the alternate URL that is used to invoke the servlet or JSP file. The property is valid only if the EdgeCacheable property also is set for the cache entry.
persist-to-disk	All	True or false. The default is `true`. When this property is set to `false`, the cache entry is not written to the disk when overflow or server stopping occurs.

Table 39. Property values (continued). Property values and valid classes.

Property	Valid classes	Value

save-attributes	Servlet and portlet	True or false. The default is `true`. When this property is set to `false`, the request attributes are not saved with the cache entry.
		
		Use the <exclude> element to specify the request attributes that do not apply to the save-attributes property. For example, to save only the attr1 attribute with the cache entry:
		`<property name=`
		`"save-attributes">false`
		`<exclude>attr1</exclude>`
		`</property>`
		
		To save all attributes except the `attr1` attribute in the cache entry, set the property to true in the preceding sample. If you do not use the <exclude> element, either all or no request attributes are saved with the cache entry.

delay-invalidations	Command	True or false. When this property is set to `true`, the commands that are invalidating cached objects based on the invalidation rules in this cache entry invalidate the cache entries after running. By default, the invalidation occurs before the command runs.

store-cookies	Servlet and portlet	On or off. The default is `On`. This property takes one or more cookie name as its argument which is saved along with the cache object and restored by the servlet cache in the response with a set-cookie header.
		
		Save all cookies except cookie1 as part of the cache-entry as follows:
		`<property name=`
		` "store-cookies">true`
		`<exclude>cookie</exclude>`
		`</property>`
		
		Save only cookie1 as part of the cache-entry, as follows:
		`<property name=`
		` "store-cookies">false`
		`<exclude><cookie1</exclude>`
		`</property>`

ignore-get-post	Servlet and portlet	True or false. The default is `false`. When the property is set to `true` the request type is not appended to the cache-id for GET and POST requests unless the requestType component requestType component subelement is defined. By default the request type is automatically appended to the cache-id for GET and POST requests.
ignore-char-encoding	Servlet and portlet	True or false. The default value is `false`. When the property is set to `true`, UTF-8 character encoding is not appended to the cache ID. Appending UTF-8 character encoding to the cache ID might lead to multiple copies of fragments, which unnecessarily increases the size of the cache.

do-not-cache	Servlet and portlet	Defines a fragment that is neither cached nor consumed by its parent.
		`<cache-entry>`
		`...`
		`<property name="do-not-cache">`
		` true</property>`
		
		or
		`<cache-id>`
		`<property name="do-not-cache">`
		` true</property>`
		`</cache-id>`
		`</cache-entry>`

cache-id

To cache an object, the application server must know how to generate a unique ID for different invocations of that object. These IDs are built either from user-written custom Java code or from rules that are defined in the cache policy of each cache entry. Each cache entry can have multiple cache ID rules that run in order until either:

- A rule returns a non-empty cache ID, or
- No more rules are left to run.

If none of the cache ID generation rules produce a valid cache ID, the object is not cached.

Each cache-id element defines a rule for caching an object and is composed of the sub-elements component, timeout, inactivity, priority, property, idgenerator, and metadatagenerator. The following example illustrates a cache-id element:

```
<cache-id>
   component*| timeout? | inactivity? | priority? | property* | idgenerator? | metadatagenerator?
</cache-id>
```

component subelement

Use the component subelement to generate a portion of the cache ID. The component subelement consists of the attributes id, type, and ignore-value, and the elements index, method, field, required, value, and not-value.

- Use the id attribute to identify the component.
- Use the type attribute to identify the type of component. The following table lists the values for the type.

gotcha: When the parameter component subelement is specified in the cachespec.xml file, the web container sets character encoding based on the encoding setting obtained from the request object. If a servlet subsequently invokes the setCharacterEncoding method, the method has not effect because character encoding cannot be changed after it has already been set.

Type	Valid classes	Meaning
method	Command	Calls the indicated method on the command or object
field	Command	Retrieves the named field in the command or object
parameter	Servlet and portlet	Retrieves the named parameter value from the request object
parameter-list	Servlet and portlet	Retrieves a list of values for the named parameter **gotcha:** The cache grabs the values for all of the parameters with this name and uses all of these values, in the order they are grabbed, to create the cache ID
session	Servlet and portlet	Retrieves the named value from the HTTP session
cookie	Servlet	Retrieves the named cookie value
attribute	Servlet and portlet	Retrieves the named request attribute
header	Servlet, web service, and portlet	Retrieves the named request header
pathInfo	Servlet	Retrieves the pathInfo element from the request
servletpath	Servlet	Retrieves the servlet path
locale	Servlet and portlet	Retrieves the request locale **Attention:** The locale component is permissible for edgecacheable entries only, when using RRD. The locale component is not valid for all other ESI versions.
requestType	Servlet and portlet	Retrieves the HTTP request method from the request. **Attention:** The requestType component is permissible for edgecacheable entries only, when using RRD. The requestType component is not valid for all other ESI versions.
tiles_attribute	Servlet and portlet	Retrieves the value of an attribute from a tile.
SOAPEnvelope	Web service and web services client cache	Retrieves the SOAPEnvelope element from a web services request. An ID attribute of Hash uses a Hash of the SOAPEnvelope element, while Literal uses the SOAPEnvelope element as received.

Type	Valid classes	Meaning
SOAPAction	Web service	Retrieves the SOAPAction header, if available, for a web services request.
serviceOperation	Web service	Retrieves the service operation for a web services request
serviceOperationParameter	Web service	Retrieves the specified parameter from a web services request
operation	Web services client cache	Indicates an operation type in the Web Services Description Language (WSDL) file. The id attribute is ignored and the value is the operation or method name. If the namespace of the operation is specified, format the value as `namespaceOfOperation:nameOfOperation`
part	Web services client cache	Indicates an input message part in the WSDL file or a request parameter. Its id attribute is the part or parameter name, and the value is the part or parameter value.

SOAPHeaderEntry	Web services client cache	Retrieves special information in the Simple Object Access Protocol (SOAP) header of the web services request. The id attribute specifies the name of the entry. In addition, the entry of the SOAP header in the SOAP request must have the actor attribute, which contains com.ibm.websphere.cache. For example:
		`<soapenv:Header>`
		`<getQuote soapenv:actor=`
		`"com.ibm.websphere.cache">IBM`
		`</getQuote>`
		`</soapenv:Header>`

portletSession	Portlet	Retrieves the named value from the portlet session
portletWindowId	Portlet	Retrieves the portlet window ID from the portlet request object
portletMode	Portlet	Retrieves the portlet mode from the portlet request object
portletWindowsState	Portlet	Retrieves the portlet window state from the portlet request object
sessionID	Servlet and portlet	Retrieves the HTTP session ID

- Use the ignore-value attribute to specify whether or not to use the value that is returned by this component in cache ID formation. This attribute is optional with a default value of `false`. If the value is `true`, only the ID of the component is used when creating a cache ID, or no output is used when creating a dependency or invalidation ID.
- Use the method element to call a void method on a returned object. You can infinitely nest method and field objects in any combination. The method must be public and is not valid for edge-cacheable components. For example:

  ```
  <component id="getUser" type="method"><method>getUserInfo
  <method>getName</method></method></component>
  ```

  This method is equivalent to getUser().getUserInfo().getName()

  For component types attribute, method, or field that can return an object, when the object returned is a collection or array, the ID is created with a comma separated list of the elements in the collection or array. For example, if the request attribute users returns an array [a, b] and the cache entry is defined like the following example:

  ```
  <cache-entry>
   <class>servlet</class>
   <name>xxx.jsp</name>
   <cache-id>
    .
  ```

```
  .
  <component id="users" type="attribute">
   <required>true</required>
  </component>
  .
  .
 </cache-id>
 <dependency-id>dep
  <component id="users" type="attribute">
   <required>true</required>
  </component>
 </dependency-id>
</cache-entry>
```

The cache id contains the string `users: a,b`. The dependency id is `dep: a,b`.

Use the multipleIDs attribute with the component types to specify and generate multiple dependency IDs (or invalidation IDs), based on the items in the collection or array. For example:

```
<cache-entry>
 <class>servlet</class>
 <name>xxx.jsp</name>
 <cache-id>
  .
  .
   <component id="users" type="attribute">
    <required>true</required>
   </component>
  .
  .
 </cache-id>
 <dependency-id>dep
  <component id="users" type="attribute" multipleIDs="true">
   <required>true</required>
  </component>
 </dependency-id>
</cache-entry>
```

The cache policy will generates the following dependency IDs:

- `dep:a,b`
- `dep:a`
- `dep:b`

Use the index element with the previous component type to add only the value of the element at the specified index position in the collection or array, to the ID that is being created.

```
<cache-entry>
 <class>servlet</class>
 <name>xxx.jsp</name>
 <cache-id>
  .
  .
   <component id="users" type="attribute">
    <required>true</required>
    <index>1</index>
   </component>
  .
  .
 </cache-id>
 <dependency-id>dep
  <component id="users" type="attribute" multipleIDs="true">
   <required>true</required>
  </component>
 </dependency-id>
</cache-entry>
```

The previous cache policy generates the following component to use in the cache ID: `users: b`.
Use the <method> element to call a void method on a returned object.

- Use the field element to access a field in a returned object. You can infinitely nest method and field objects in any combination. The field must be public. This field is not valid for edge-cacheable components. For example:

```
<component id="getUser" type="method"><method>getUserInfo
<field>name</field></method></component>
```

This method is equivalent to the getUser().getUserInfo().name method.

- Use the required element to specify whether or not this component must return a non-null value for this cache ID to represent a valid cache. If set to `true`, this component must return a non-null value for this cache ID to represent a valid cache ID. If set to `false`, the default, a non-null value is used in the formation of the cache ID and a null value means that this component is not used at all in the ID formation. For example:

  ```
  <required>true</required>
  ```
- Use the value element to specify values that must match to use this component in cache ID formation. For example:

  ```
  <component id="getUser" type="method"><value>blue</value>
  <value>red</value> </component>
  ```
- Use the not-value element to specify values that must not match to use this component in cache ID formation. This method is similar to value element, but instead prescribes the defined values from caching. You can use multiple not-value elements when more than one value that is not valid exists. For example:

  ```
  <component id="getUser" type="method">
  <required>true</required>
  <not-value>blue</not-value>
  <not-value>red</not-value></component>
  ```

The component subelement can have either a method and a field element, a value element, or a not-value element. The method and field elements apply to commands only. The following example illustrates the attributes of a component sub-element:

```
<component id="isValid" type="method" ignore-value="true"><component>
```

timeout subelement

The timeout subelement is used to specify an absolute time-to-live (TTL) value for the cache entry. For example,

```
<timeout>value</timeout>
```

where *value* is the amount of time, in seconds, to keep the cache entry. Cache entries that are in memory are kept indefinitely, as long as the entries remain in memory. Cache entries that are stored on disk are evicted if they are not accessed for 24 hours.

inactivity subelement

The inactivity subelement is used to specify a time-to-live (TTL) value for the cache entry based on the last time that the cache entry was accessed. It is a subelement of the cache-id element.

```
<inactivity>value</inactivity>
```

where *value* is the amount of time, in seconds, to keep the cache entry in the cache after the last cache hit.

priority subelement

Use the priority subelement to specify the priority of a cache entry in a cache. The priority weighting is used by the least recently used (LRU) algorithm of the cache to decide which entries to remove from the cache if the cache runs out of storage space. For example,

```
<priority>value</priority>
```

where *value* is a positive integer between 1 and 16 inclusive.

Samples

The following sample keeps the cache entry in the cache for a minimum of 35 seconds and a maximum of 180 seconds. If the cache entry is accessed within each 35 second inactivity period, the inactivity period is extended for another 35 seconds. However, because the timeout element is also configured, the cache entry is always invalidated after 180 seconds. If the cache entry is not accessed within the 35 second period, the entry is removed from the cache.

```
<cache-id>
 <component id="timeout" type="parameter">
  <required>true</required>
 </component>
```

```
 <timeout>180</timeout>
 <inactivity>35</inactivity>
 <priority>1</priority>
</cache-id>
```

The following sample keeps the cache entry in the cache for a minimum of 600 seconds. If the cache entry is accessed within each 600 second period, the inactivity period is extended for another 600 seconds. If the cache entry is not accessed within the 600 second period, the cache entry is removed from the cache.

```
<cache-id>
 <component id="timeout" type="parameter">
  <required>true</required>
 </component>
 <inactivity>600</inactivity>
 <priority>1</priority>
</cache-id>
```

In the following sample, the value for inactivity has no meaning because the timeout period is less than the inactivity period. The cache entry is always invalidated after 180 seconds, no matter how often the cache entry is accessed.

```
<cache-id>
 <component id="timeout" type="parameter">
  <required>true</required>
 </component>
 <timeout>180</timeout>
 <inactivity>600</inactivity>
 <priority>1</priority>
</cache-id>
```

property subelement

Use the property subelement to specify generic properties for the cache entry. For example,

```
<property name="key">value</property>
```

where *key* is the name of the property to define, and *value* is the corresponding value.

For example:

```
<property name="description">The Snoop Servlet</property>
```

Table 40. Property subelement values.. Property valid classes and meaning.

Property	Valid classes	Meaning
sharing-policy/timeout/priority	All	Overrides the settings for the containing cache entry when the request matches this cache ID.
EdgeCacheable	Servlet	Overrides the settings for the containing cache entry when the request matches this cache ID.

idgenerator and metadatagenerator sub-elements

Use the idgenerator element to specify the class name that is loaded for the generation of the cache ID. The IdGenerator element must implement the com.ibm.websphere.servlet.cache.IdGenerator interface for a servlet or the com.ibm.websphere.webservices.IdGenerator interface for the web services client cache. An example of the idgenerator element follows:

```
<idgenerator> class name </idgenerator>
```

Where *class name* is the fully-qualified name of the class to use. Define this generator class in a shared library.

Use the metadatagenerator element inside the cache-id element to specify the class name loaded for the metadata generation. The MetadataGenerator class must implement the com.ibm.websphere.servlet.cache.MetaDataGenerator interface for a servlet or the

com.ibm.websphere.cache.webservices.MetaDataGenerator interface for a web services client cache. The MetadataGenerator class defines properties like timeout, inactivity, external caching properties or dependencies. An example of the `metadatagenerator` element follows:

```
<metadatagenerator> classname </metadatagenerator>
```

In this example, class name is the fully-qualified name of the class to use. Define this generator class in a shared library.

dependency-id element

Use the dependency-id element to specify additional cache identifiers that associate multiple cache entries to the same group identifier.

The value of the dependency-id element is generated by concatenating the dependency ID base string with the values that are returned by its component elements. If a required component returns a null value, the entire dependency does not generate and is not used. Validate the dependency IDs explicitly through the dynamic cache API, or use the invalidation element. Multiple dependency ID rules can exist in one cache-entry element. All dependency rules run separately.

invalidation element

To invalidate cached objects, the application server must generate unique invalidation IDs. Build invalidation IDs by writing custom Java code or through rules that are defined in the cache policy of each cache entry. The following example illustrates an invalidation in the cache policy:

```
<invalidation>component* | invalidationgenerator? </invalidation>
```

invalidationgenerator subelement

The invalidationgenerator element is used with the web services client cache only. Use the invalidationgenerator element to specify the class name to load for generating invalidation IDs. The InvalidationGenerator class must implement the com.ibm.websphere.cache.webservices.InvalidationGenerator interface. An example of the invalidationgenerator element follows:

```
<invalidationgenerator>class name</invalidationgenerator>
```

In this example, classname is the fully qualified name of the class that implements the com.ibm.websphere.cache.webservices.InvalidationGenerator interface. Define this generator class in a shared library.

Example: Configuring the dynamic cache service

This example puts all of the steps together for configuring the dynamic cache service with the `cachespec.xml` file, showing the use of the cache ID generation rules, dependency IDs, and invalidation rules.

Suppose that a servlet manages a simple news site. This servlet uses the query parameter "action" to determine if the request views (query parameter "view") news or updates (query parameter "update") news (used by the administrator). Another query parameter "category" selects the news category. Suppose that this site supports an optional customized layout that is stored in the user's session using the attribute name "layout". Here are example URL requests to this servlet:

`http://yourhost/yourwebapp/newscontroller?action=view&category=sports` (Returns a news page for the sports category)

`http://yourhost/yourwebapp/newscontroller?action=view&category=money` (Returns a news page for the money category)

`http://yourhost/yourwebapp/newscontroller?action=update&category=fashion` (Allows the administrator to update news in the fashion category)

Here are the steps for configuring the dynamic cache service for this example with the `cachespec.xml` file:

1. Define the <cache-entry> elements that are necessary to identify the servlet. In this case, the URI for the servlet is "newscontroller", so this is the cache-entry <name> element. Because this example caches a servlet or JavaServer Pages (JSP) file, the cache entry class is "servlet".

```
<cache-entry>
<name> /newscontroller </name>
<class>servlet  </class>
 </cache-entry>
```

2. Define cache ID generation rules. This servlet caches only when action=view, so one component of the cache ID is the parameter "action" when the value equals "view". The news category is also an essential part of the cache ID. The optional session attribute for the user's layout is included in the cache ID. The cache entry is now:

```
<cache-entry>
 <name> /newscontroller </name>
 <class>servlet  </class>
  <cache-id>
  <component id="action" type="parameter">
   <value>view</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
  </component>
  <component id="layout" type="session">
   <required>false</required>
  </component>
 </cache-id>
</cache-entry>
```

3. Define dependency ID rules. For this servlet, a dependency ID is added for the category. Later, when the category is invalidated due to an update event, all views of that news category are invalidated. Following is an example of the cache entry after adding the dependency ID:

```
<cache-entry>
 <name>newscontroller </name>
 <class>servlet  </class>
  <cache-id>
  <component id="action" type="parameter">
   <value>view</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
  </component>
  <component id="layout" type="session">
   <required>false</required>
  </component>
 </cache-id>
 <dependency-id>category
  <component id="category" type="parameter">
   <required>true</required>
  </component>
 </dependency-id>
</cache-entry>
```

4. Define invalidation rules. Because a category dependency ID is already defined, define an invalidation rule to invalidate the category when action=update. To incorporate the conditional logic, add

"ignore-value" components into the invalidation rule. These components do not add to the output of the invalidation ID, but only determine whether or not the invalidation ID creates and runs. The final cache-entry now looks like the following:

```
<cache-entry>
 <name>newscontroller </name>
 <class>servlet  </class>
  <cache-id>
  <component id="action" type="parameter">
   <value>view</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
  </component>
  <component id="layout" type="session">
   <required>false</required>
  </component>
 </cache-id>
 <dependency-id>category
  <component id="category" type="parameter">
   <required>true</required>
  </component>
 </dependency-id>
 <invalidation>category
  <component id="action" type="parameter" ignore-value="true">
   <value>update</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
     </component>
 </invalidation>
</cache-entry>
```

cacheinstances.properties file

Use the information in this document as a reference of the names, values, and explanations that you can use in the cacheinstances.properties file.

The following list provides the property names, associated values, and explanations for the cacheinstance.properties file.

Property name - x is the instance number, which starts with 0	**Version**	**Scope**	**Possible value**	**Description**
Cache core properties				
`cache.instance.x`	5.1.x and later	Per cache instance	any string (no default set)	Specifies cache instance name or JNDI name.
`cache.instance.x. cacheSize`	5.1.x and later	Per cache instance	> 0 (default=2000)	Specifies the maximum number of entries that are held in memory cache.
`cache.instance.x. disableDependencyId`	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies that the dynamic cache service supports cache entry dependency IDs. Disable this option if you do not need to use dependency IDs. Dependency IDs specify additional cache group identifiers that associate multiple cache entries to the same group identifier in your cache policy.

Property name - x is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. disableTemplates Support	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether template support feature is enabled.
cache.instance.x. useListenerContext	5.1.x and later	Per cache instance	True or false (default=false)	Set this value to true to have invalidation events sent to registered invalidation listeners, using the Java Platform, Enterprise Edition (Java EE) context of the listener. If you want to use listener Java EE context for callback, set this value to true. If you want to use the caller thread context for callback, set this value to false.
cache.instance.x. enableNioSupport	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether DistributedMap or DistributedNioMap is used.
cache.instance.x. memoryCacheSizeInMB	7.0	Per cache instance	> 0 (default: -1 limit does not exist)	Specifies a value for the maximum memory cache size in megabytes (MB)
cache.instance.x. memoryCacheHigh Threshold	7.0	Per cache instance	> 0 % (default=95)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the memory cache size in MB. The higher value is used when limit memory cache size in MB is specified.
cache.instance.x. memoryCacheLow Threshold	7.0	Per cache instance	> 0 % (default=80)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the memory cache size in MB. The lesser value is used when limit memory cache size in MB is specified.
cache.instance.x. createCacheAt ServerStartup	7.0	Per cache instance	True or false (default=false)	Specifies whether the configured cache instance is created during the server startup. This is useful when cache replication feature is used. However, the time for server startup will take long.
Cache servlet/JavaServer Pages (JSP) caching properties				
cache.instance.x. cascadeCachespec Properties	6.0.2.19, 6.1.0.9 and later	Per cache instance	True or false (default=false)	A configurable change in the behavior of the cache so that the child pages and fragments inherit the cache specification properties of their parent pages and fragments. If the request for a fragment does not match a defined cache policy, the fragment will inherit the save-attributes and the store-cookies properties from its parent fragment. Enable this cascade of save-attributes and store-cookies properties by setting the value to true.

Property name - x is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. disableStoreCookies	6.0.2.9, 6.1.x and later	Per cache instance	"none", "ALL","All", cache instance name, comma delineated list of cookie names, (default="none")	Specifies whether disable store cookies is NONE or ALL. Stores cookies as part of the response by default unless configured otherwise on a per request basis in cachespec.xml file. There is a risk of sharing cookies between users, which violates security.
cache.instance.x. enableServlet Support	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether the cache instance is servlet cache or object cache.
Cache disk offload properties				
cache.instance.x. enableDiskOffload	5.1.x and later	Per cache instance	True or false (default=false)	Specifies whether disk offload is enabled.
cache.instance.x. diskOffload Location	5.1.x and later	Per cache instance	String – For example: . $(app_server_root) /diskOffload	Specifies the location on the disk to save cache entries when disk offload is enabled.
cache.instance.x. diskCacheSize	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	>= 0 (0=limit does not exist)	Specifies a value for the maximum disk cache size in number of entries.
cache.instance.x. diskCacheSizeInGB	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	0 or > 2 in GB (0=limit does not exist)	Specifies a value for the maximum disk cache size in gigabytes (GB).
cache.instance.x. diskCacheEntrySize InMB	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	>= 0 in MB (0=limit does not exist)	Specifies a value for the maximum size of an individual cache entry in megabytes (MB). Any cache entry that is larger than this, when evicted from memory, will not be offloaded to disk.
cache.instance.x. flushToDiskOnStop	5.1.x and later	Per cache instance	True or false (default = false)	Specifies if in-memory cached objects are saved to disk when the server stops.
cache.instance.x. diskCache PerformanceLevel	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	0=low 1=balance 2=custom 3=high (default=1)	Specifies the performance level to tune the performance of the disk cache.
cache.instance.x. htodCleanup Frequency	5.1.1.2 and later	Per cache instance	0 <= x <= 1440 in minutes (0=cleanup at midnight)	Specifies a value for the disk cache cleanup frequency, in minutes. If this value is set to 0, the cleanup runs only at midnight. This setting applies only when the Disk Offload Performance Level is low, balanced, or custom. The high performance level does not require disk cleanup, and this value is ignored.
cache.instance.x. htodDelayOffload DepIdBuckets	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 (default=1000)	Specifies a value for the maximum number of dependency identifier buckets in the disk cache metadata in memory. If this limit is exceeded, the information is offloaded to the disk. This setting applies only when the disk cache performance level is custom.

Property name - x is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x.htodDelayOffloadTemplateBuckets	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 (default=100)	Specifies a value for the maximum number of template buckets that are in the disk cache metadata in memory. If this limit is exceeded, the information is offloaded to the disk. This setting applies only when the disk cache performance level is custom.
cache.instance.x htodDelayOffloadEntriesLimit	5.1.1.2 and later	Per cache instance	> 0 (default=1000)	Specifies a value for the maximum number of cache identifiers that are stored for an individual dependency ID or template in the disk cache metadata in memory. If this limit is exceeded, the information is offloaded to the disk. This setting applies only when the disk offload performance level is custom.
cache.instance.x.diskCacheEvictionPolicy	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	0=disable 1=random 2:size (default=0)	Specifies the eviction algorithm that the disk cache will use to evict entries once the high threshold is reached.
cache.instance.x.diskCacheHighThreshold	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 % (default=80)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the disk cache size in GB or entries. The high value is used when limit disk cache size in GB and limit disk cache size in entries are specified.
cache.instance.x.diskCacheLowThreshold	5.1.1.13, 6.0.2.17, 6.1.x and later	Per cache instance	> 0 % (default=70)	Specifies when the eviction policy runs. The threshold is expressed in terms of the percentage of the disk cache size in GB or entries. The lesser value is used when limit disk cache size in GB and limit disk cache size in entries are specified.
Cache replication properties				
cache.instance.x.enableCacheReplication	6.0.2.x and later	Per cache instance	True or false (default=false)	Specifies whether cache replication is enabled. Use cache replication to have cache entries copied to multiple application servers configured in the same replication domain.
cache.instance.x.replicationType	5.1.x and later	Per cache instance	1 (Not shared, 2 (Push), 4 (Push and pull)	Specifies the global sharing policy for this application server.
cache.instance.x.replicationDomain	6.0.2.x and later	Per cache instance	String – For example: DynamicCacheDomain	Specifies a replication domain from which your data is replicated.
cache.instance.x.useServerClassLoader	5.1.1.9, 6.0.2.9, 6.1.x and later	Per cache instance	True or false (default=false)	Specifies whether using server class loader is enabled. Setting this value to true, deserializes the InvalidationEvent using system classloader first and then using application classloader, if that fails. This improves performance.

Property name - x is the instance number, which starts with 0	Version	Scope	Possible value	Description
cache.instance.x. cacheEntryWindow	5.1.1.13, 6.0.2.17, 6.1.0.7 and later	Per cache instance	> 0 (default=50)	Specifies a limit on the total number of cache entries that are sent by the data replication service (DRS) in terms of number of entries.
cache.instance.x. cachePercentage Window	5.1.1.13, 6.0.2.17, 6.1.0.7 and later	Per cache instance	> 0 % (default=2)	Specifies a limit on the number of cache entries that are sent by DRS in terms of the percentage of total cache in memory.
cache.instance.x. cacheInvalidate EntryWindow	5.1.1.14, 6.0.2.19, 6.1.0.7 and later	Per cache instance	> 0 (default=50)	Specifies a limit on the total number of invalidation events that are sent by DRS in terms of number of entries.
cache.instance.x. cacheInvalidate PercentWindow	5.1.1.14, 6.0.2.19, 6.1.0.7 and later	Per cache instance	> 0 % (default=2)	Specifies a limit on the number of invalidation events that are sent by DRS in terms of the percentage of total cache in memory.
cache.instance.x. filterTimeOut Invalidation	6.0.2.13, 6.1.x and later	Per cache instance	True or false (default=false)	Specifies whether sending invalidations that are based on timeout eviction is enabled.
cache.instance.x. filterLRU Invalidation	6.0.2.13, 6.1.x and later	Per cache instance	True or false (default=false)	Specifies whether sending invalidations that are based on LRU eviction is enabled.
cache.instance.x. ignoreValueIn InvalidationEvent	5.1.1.13, 6.0.2.17, 6.1.x or later	Per cache instance	True or false (default=false)	Specifies whether the cache value of Invalidation event is ignored. If it is true, the cache value of Invalidation event is set to NULL when the code is returned to the caller.

Chapter 9. Developing Dynamic and EJB query

This page provides a starting point for finding information about dynamic query, a WebSphere programming extension for unprecedented application flexibility. This information also includes Enterprise JavaBeans (EJB) query, the Java feature upon which the WebSphere extension is built.

Dynamic query lets you dynamically build and submit queries that select, sort, join, and perform calculations on application data at run time.

Dynamic query service provides the ability to pass in and process EJB query language queries at run time, eliminating the need to hard-code required queries into deployment descriptors during application development.

Developing applications that use EJB query

EJB query language

EJB query language enables you to write queries based on entity beans without knowing the underlying relational schema.

An EJB query is a string that contains the following elements:
- a SELECT clause that specifies the enterprise beans or values to return;
- a FROM clause that names the bean collections;
- an optional WHERE clause that contains search predicates over the collections;
- an optional GROUP BY and HAVING clause; see the topic Aggregation functions.
- an optional ORDER BY clause that specifies the ordering of the result collection.

Collections of entity beans are identified in EJB queries through the use of their abstract schema name in the query FROM clause.

The elements of EJB query language are discussed in more detail in the following related topics.

FROM clause

The FROM clause specifies the collections of objects to which the query is to be applied. Each collection is specified either by an abstract schema name (ASN) or by a path expression identifying a relationship. An identification variable is defined for each collection.

Conceptually, the semantics of the query is to form a temporary collection of tuples, `R`, with elements consisting of all possible combinations of objects from the collections. This collection is subject to the constraints imposed by any path relationships and by the JOIN operation. The JOIN can be either an *inner* or *outer* join.

The identification variables are bound to elements of the tuple. After forming the temporary collection, the search conditions of the WHERE clause are applied to R, and yield a new temporary collection, `R1`. The ORDER BY, GROUP BY, HAVING, and SELECT clauses are applied to R1 to yield the final result.

```
from_clause::=FROM identification_variable_declaration [, {identification_variable_declaration |
collection_member_declaration } ]*

identification_variable_declaration::= range_variable_declaration [join]*

join :=  [ { LEFT [OUTER] | INNER  }]  JOIN  {collection_valued_path_expression | single_valued_path_expression}
 [AS] identifier
```

Examples: Joining collections

`DeptBean` contains records 10, 20, and 30. `EmpBean` contains records 1, 2, and 3 that are related to department 10, and records 4 and 5 that are related to department 20. Department 30 has no employees.

```
SELECT d FROM  DeptBean AS d, EmpBean AS e
WHERE d.name = e.name
```

The comma syntax performs an inner join resulting in all possible combinations. In this example, R would consist of 15 tuples (3 departments x 5 employees). If any collection is empty, then R is also empty. The keyword AS is optional.

This example shows that a collection can be joined with itself.

```
SELECT d FROM DeptBean AS d, DeptBean AS d1
```

R would consist of 9 tuples (3 departments x 3 departments).

Examples: Relationship joins

A collection can be a relationship based on a previously declared identifier as in

```
SELECT e FROM DeptBean AS d , IN (d.emps) AS e
```

R would contain 5 tuples. Department 30 would not appear in R because it contains no employees. Department 10 would appear in 3 tuples and department 20 would appear in 2 tuples. IN can only refer to multi-valued relationships. The following is not valid

```
SELECT m FROM EmpBean e,  IN( e.dept.mgr) as m   INVALID
```

When joining with a relationship the alternate syntax INNER JOIN (keyword INNER is optional) can also be used, as shown here.

```
SELECT e FROM DeptBean AS d INNER JOIN d.emps AS e
```

An ASN declaration (d in the previous query) can be followed by one or more join clauses. The relationship following the JOIN keyword must be related (directly or indirectly) to the ASN declaration. Unlike the case with the IN clause, relationships used in a join clause can be single- or multi-valued. This query has the same semantics as the query

```
SELECT e FROM DeptBean AS d , IN (d.emps) AS e
```

You can use multiple joins together.

```
SELECT m FROM EmpBean e JOIN e.dept d JOIN d.mgr m
```

This is equivalent to

```
SELECT m FROM EmpBean e JOIN e.dept.mgr m
```

Examples: OUTER JOIN

An OUTER JOIN results in a temporary collection that contains combinations of the *left* and *right* operands, subject to the relationship constraints and such that the left operand always appears in R. In the example an outer join results in a temporary collection R that contains department 30, even though the collection d.emps is empty. The tuple contains Department 30 along with a NULL value. References to e in the query yields a null value.

```
SELECT e FROM DeptBean AS d LEFT OUTER JOIN d.emps AS e
```

The keyword OUTER is optional, as shown here.

```
SELECT e FROM DeptBean AS d LEFT JOIN d.emps AS e
```

You can also use combinations of INNER and OUTER JOIN.

```
SELECT m FROM EmpBean e JOIN e.dept d LEFT JOIN d.mgr m
```

Inheritance in EJB query

If an Enterprise JavaBeans (EJB) inheritance hierarchy has been defined for an abstract schema, using the abstract schema name in a query statement implies the collection of objects for that abstract schema as well as all subtypes.

Example: Inheritance

Suppose that bean `ManagerBean` is defined as a subtype of `EmpBean` and `ExecutiveBean` is a subtype of `ManagerBean` in an EJB inheritance hierarchy. The following query returns employees as well as managers and executives:

```
SELECT OBJECT(e) FROM EmpBean e
```

Path expressions

A path expression is an identification variable followed by the navigation operator (.) and a container managed persistence (CMP) or relationship name.

A path expression that leads to a cmr field can be further navigated if the cmr field is single-valued. If the path expression leads to a multi-valued relationship, then the path expression is terminal and cannot be further navigated. If the path expression leads to a CMP field whose type is a value object, it is possible to navigate to attributes of the value object.

Example: Value object

Assume that `address` is a CMP field for `EmpBean`, which is a value object.

```
SELECT  object(e)  FROM EmpBean e
WHERE  e.address.distance('San Jose') < 10  and e.address.zip = 95037
```

It is best to use the composer pattern to map value object attributes to relational columns if you intend to search on value attributes. If you store value objects in serialized format, then each value object must be retrieved from the database and deserialized. Value object methods can only be done in dynamic queries.

A path expression can also navigate to a bean method. The method must be defined on either the remote or local bean interface. Methods can only be used in dynamic queries. You cannot mix both remote and local methods in a single query statement.

If the query contains remote methods, the dynamic query must be executed using the query remote interface. Using the query remote interface causes the query service to activate beans and create instances of the remote bean interface

Likewise, a query statement with local bean methods must be executed with the query local interface. This causes the query service to activate beans and local interface instances.

Do not use get methods to access CMP and cmr fields of a bean.

If a method has overloaded definitions, the overloaded methods must have different number of parameters.

Methods must have non-void return types and method arguments and return types must be either primitive types byte, short, int, long, float, double, boolean, char or wrapper types from the following list:

Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Date

If any input argument to a method is NULL, it is assumed the method returns a NULL value and the method is not invoked.

A collection valued path expression can be used in the FROM clause as a collection member declaration, and with the IS EMPTY, MEMBER OF, and EXISTS predicates in the WHERE clause.

Table 41. FROM clause usage. The following table lists three valid From clause usage and one invalid usage.

FROM clause usage statement	Validity statement
`FROM EmpBean e WHERE e.dept.mgr.name='Bob'`	OK
`FROM EmpBean e WHERE e.dept.emps.name='BOB'`	INVALID -- cannot navigate through emps because it is multivalued
`FROM EmpBean e, IN (e.dept.emps) e1 WHERE e1.name='BOB'`	OK
`FROM EmpBean e WHERE e.dept.emps IS EMPTY`	OK

WHERE clause

The WHERE clause lists search conditions for items to add to a result set.

The WHERE clause contains search conditions composed of the following:
- literal values
- input parameters
- expressions
- basic predicates
- quantified predicates
- BETWEEN predicate
- IN predicate
- LIKE predicate
- NULL predicate
- EMPTY collection predicate
- MEMBER OF predicate
- EXISTS predicate
- IS OF TYPE predicate

If the search condition evaluates to TRUE, the tuple is added to the result set.

Literals:

Literals can be considered constants that do not change in value.

A string literal is enclosed in single quotes. A single quote that occurs within a string literal is represented by two single quotes. For example: 'Tom''s'. A string literal cannot exceed the maximum length that is supported by the underlying persistent datastore.

A numeric literal can be any of the following:
- an exact value such as 57, -957, +66
- any value supported by Java long
- a decimal literal such as 57.5, -47.02
- an approximate numeric value such as 7E3, -57.4E-2

A decimal or approximate numeric value must be in the range supported by the underlying persistent datastore.

A boolean literal can be the keyword TRUE or FALSE and is case insensitive.

Input parameters:

Input parameters are designated by the question mark followed by a number; for example: `?2`. Input parameters are numbered starting at 1 and correspond to the arguments of the finder or select method; therefore, a query must not contain an input parameter that exceeds the number of input arguments.

An input parameter can be a primitive type of byte, short, int, long, float, double, boolean, char or wrapper types of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Char, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, an EJBObject, or a binary data string in the form of Java byte[].

An input parameter must not have a NULL value. To search for the occurrence of a NULL value the NULL predicate should be used.

WebSphere Application Server, Expressions:

An expression specifies a value.

Conditional expressions can consist of comparison operators and logical operators (AND, OR, NOT).

Arithmetic expressions can be used in comparison expressions and can be composed of arithmetic operations and functions, path expressions that evaluate to a numeric value and numeric literals and numeric input parameters.

String expressions can be used in comparison expressions and can be composed of string functions, path expressions that evaluate to a string value and string literals and string input parameters. A CMP field of type char is handled as if it were a string of length 1.

Binary expressions can be used in comparison expressions and can be composed of path expressions that evaluate to the Java byte[] type as well as input parameters of type byte[].

Boolean expressions can be used with = and <> comparison and can be composed of path expressions that evaluate to a boolean value and TRUE and FALSE keywords and boolean input parameters.

Reference expressions can be used with = and <> comparison and can be composed of path expressions that evaluate to a cmr field, an identification variable and an input parameter whose type is an EJB reference

Four different expression types are supported for working with date-time types. For portability the java.util.Calendar type should be used. DB2 style date, time and timestamp expressions are supported if the datastore is DB2 and the CMP field is of type java.util.Date, java.sql.Date, java.sql.Time or java.sql.Timestamp. If you use DB2 UDB, you might obtain a syntax error when using the java.sql.Timestamp.ojbect. You must use the syntax TIMESTAMP 'yyyy-mm-dd hh:mm:ss.nnnn'.

A Calendar type can be compared to another Calendar type, an exact numeric literal or input parameter of type long whose value is the standard Java long millisecond value.

The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate <  631180800232
```

Date expressions can be used in comparison expressions and can be composed of operators + - , date duration expressions and date functions, path expressions that evaluate to a date value, string representation of a date and date input parameters.

Time expressions can be used in comparison expressions and can be composed of operators + - , time duration expressions and time functions, path expressions that evaluate to a time value, string representation of time and time input parameters.

Timestamp expressions can be used in comparison expressions and can be composed of operators + - , timestamp duration expressions and timestamp functions, path expressions that evaluate to a timestamp value, string representation of a timestamp and timestamp input parameters.

Standard bracketing ( ) for ordering expression evaluation is supported.

The operators and their precedence order from highest to lowest are:
- Navigation operator (.)
- Arithmetic operators in precedence order:
  - + - unary
  - * / multiply, divide
  - + - add, subtract
- Comparison operators: =, >, <, >=, <=, <>(not equal)
- Logical operator NOT
- Logical operator AND
- Logical operator OR

Null value semantics:

The following describe the semantics of NULL values.
- Comparison or arithmetic operations with an unknown (NULL) value yield an unknown value
- In a Java 2 platform, Enterprise Edition (J2EE) version 1.3 application, a path expression uses an outer-join semantic where a NULL field or cmr value evaluates to NULL. In J2EE version 1.4, the path expression uses an inner-join semantic.
- The IS NULL and IS NOT NULL operators can be applied to path expressions and return TRUE or FALSE. Boolean operators AND, OR and NOT use three valued logic.

Table 42. Null value semantics. The following table describes the semantics of NULL values.

AND	**True**	**False**	**Unknown**
`True`	True	False	Unknown
`False`	False	False	False
`Unknown`	Unknown	False	Unknown

Table 43. Null value semantics. The following table describes the semantics of NULL values.

OR	**True**	**False**	**Unknown**
`True`	True	True	True
`False`	True	False	Unknown
`Unknown`	True	Unknown	Unknown

Table 44. Null value semantics. The following table describes the semantics of NULL values.

	NOT
`True`	False
`False`	True
`Unknown`	Unknown

Example: Null value semantics

```
select object(e) from EmpBean where e.salary > 10  and e.dept.budget > 100
```

If salary is NULL the evaluation of `e.salary > 10` returns unknown and the employee object is not returned. If the cmr field dept or budget is NULL evaluation of `e.dept.budget > 100` returns unknown and the employee object is not returned.

```
select object(e) from EmpBean where e.dept.budget is null
```

In J2EE 1.3 if dept or budget is NULL evaluation of `e.dept.budget` is null returns TRUE and the employee object is returned. In J2EE 1.4 the employee object is returned only if budget is NULL.

```
select object(e) from EmpBean e ,  in (e.dept.emps) e1 where e1.salary > 10
```

If dept is NULL, then the multivalued path expression `e.dept.emps` results in an empty collection (not a collection that contains a NULL value). An employee with a null dept value will not be returned.

```
select object(e) from EmpBean e where  e.dept.emps is empty
```

If dept is NULL the evaluation of the predicate in unknown and the employee object is not returned.

```
select object(e) from EmpBean e , EmpBean e1  where e member of  e1.dept.emps
```

If dept is NULL evaluation of the member of predicate returns unknown and the employee is not returned.

Date time arithmetic and comparisons:

DATE, TIME and TIMESTAMP values can be compared with another value of the same type. Comparisons are chronological. Date time values can also be incremented, decremented, and subtracted.

If the datastore is DB2, then DB2 string representation of DATE, TIME and TIMESTAMP types can also be used.

Table 45. Date, time and timestap formats. A string representation of a date or time can use ISO, USA, EUR or JIS format. A string representation of a timestamp uses ISO format.

Format	Date format	Date examples	Time format	Time examples
ISO	yyyy-mm-dd	1987-02-24 1987-2-24	hh.mm.ss	13.50.00 13.50
USA	mm/dd/yyyy	2/24/1987	hh:mm AM or PM	1:50 pm 02:10 AM
EUR	dd.mm.yyyy	24.02.1987 24.2.1987	hh.mm.ss	13.50.00 13.55
JIS	yyyy-mm-dd	1987-02-24	hh:mm:ss	13:50 13:50:05

Example 1: Date time arithmetic comparisons

```
e.hiredate > '1990-02-24'
```

The timestamp of February 24th, 1990 1:50 pm can be represented as follows:

```
'1990-02-24-13.50.00.000000'  or
'1990-02-24-13.50.00'
```

If the datastore is DB2, DB2 decimal durations can be used in expressions and comparisons. A date duration is a decimal(8,0) number that represents the difference between two dates in the format YYYYMMDD. A time duration is a decimal(6,0) number that represents the difference between two time values as HHMMSS. A timestamp duration is a decimal(20,6) number representing the differences between two timestamp values as YYYYMMDDHHMMSS.ZZZZZZ (ZZZZZZ is the number of microseconds following the decimal point).

Two date values (or time values or timestamp values) can be subtracted to yield a duration. If the second operand is greater than the first the duration is a negative decimal number. A duration can be added or subtracted from a datetime value to yield a new datetime value.

Example 2: Date time arithmetic comparisons

`DATE('3/15/2000') - '12/31/1999'` results in a decimal number 215 which is a duration of 0 years, 2 months and 15 days.

Durations are really decimal numbers and can be used in arithmetic expressions and comparisons.

`( DATE('3/15/2000') - '12/31/1999' ) + 14 > 215` evaluates to TRUE.

`DATE('12/31/1999') + DECIMAL(215,8,0)` results in a date value 3/15/2000.

`TIME('11:02:26') - '00:32:56'` results in a decimal number 102930 which is a time duration of 10 hours, 29 minutes and 30 seconds.

`TIME('00:32:56') + DECIMAL(102930,6,0)` results in a time value of 11:02:26.

`TIME('00:00:59') + DECIMAL(240000,6,0)` results in a time value of 00:00:59.

`e.hiredate + DECIMAL(500,8,0) > '2000-10-01'` means compare the hiredate plus 5 months to the date 10/01/2000.

Basic predicates:

A basic predicate compares two values.

Basic predicates can be of two forms, for example:

```
expression-1  comparison-operator  expression-2
expression-3  comparison-operator ( subselect )
```

The subselect must not return more than one value and the subselect cannot return a type of an Enterprise JavaBeans (EJB) reference. Boolean types and reference types only support = and <> comparisons.

Example: Basic predicates

```
d.name='Java Development'
e.salary > 20000
e.salary > ( select avg(e.salary) from EmpBean e)
```

Quantified predicates:

A quantified predicate compares a value with a set of values produced by a subselect.

Use the syntax:

```
expression   comparison-operator   SOME | ANY |  ALL    ( subselect )
```

The expression must not evaluate to a reference type.

When SOME or ANY is specified the result of the predicate is as follows:
- TRUE if the comparison is true for at least one value returned by the subselect.
- FALSE if the subselect is empty or if the comparison is false for every value returned by the subselect.
- UNKNOWN if the comparison is not true for all of the values returned by the subselect and at least one of the comparisons is unknown because of a null value.

When ALL is specified the result of the predicate is as follows:
- TRUE if the subselect returns empty or if the comparison is true to every value returned by the subselect.

- FALSE if the comparison is false for at least one value returned by the subselect.
- UNKNOWN if the comparison is not false for all values returned by the subselect and at least one comparison is unknown because of a null value.

BETWEEN predicate:

The BETWEEN predicate determines whether a given value lies between two other given values.

The syntax for the predicate is:

```
expression  [NOT]  BETWEEN expression-2  AND expression-3
```

The expression must not evaluate to a boolean or reference type.

Example: BETWEEN predicate

```
e.salary BETWEEN 50000 AND 60000
```

is equivalent to:

```
e.salary >= 50000  AND e.salary <= 60000
e.name NOT BETWEEN 'A' AND 'B'
```

is equivalent to:

```
e.name < 'A'  OR e.name > 'B'
```

IN predicate:

The IN predicate compares a value to a set of values.

It can have one of two forms:

```
expression [NOT] IN  ( subselect )
expression [NOT] IN  ( value1, value2,  .... )
```

ValueN can either be a literal value or an input parameter. The expression cannot evaluate to a reference type.

Example: IN predicate

```
e.salary IN ( 10000, 15000 )
```

is equivalent to

```
( e.salary = 10000  OR  e.salary = 15000 )
e.salary IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary = ANY   ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
e.salary NOT IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary <> ALL    ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

LIKE predicate:

The LIKE predicate searches a string value for a certain pattern.

The syntax for this predicate is:

```
string-expression   [NOT]  LIKE  pattern    [ ESCAPE escape-character ]
```

The pattern value is a string literal or parameter marker of type string in which the underscore (_) stands for any single character and percent (%) stands for any sequence of characters (including empty sequence). Any other character stands for itself. The escape character can be used to search for character _ and %. The escape character can be specified as a string literal or an input parameter.

If the string-expression is null, then the result is unknown.

If both string-expression and pattern are empty, then the result is true.

Example: LIKE predicate
- "`LIKE`" is true
- "`LIKE %`" is true
- `e.name LIKE 12%3` is true for "123" "12993" and false for "1234"
- `e.name LIKE 's_me'` is true for "some" and "same", false for "soome"
- `e.name LIKE '/_foo' escape '/'` is true for "_foo", false for "afoo"
- `e.name LIKE '//_foo' escape '/'` is true for "/afoo" and for "/bfoo"
- `e.name LIKE '///_foo' escape '/'` is true for "/_foo" but false for "/afoo"

NULL predicate:

The NULL predicate tests for null values.

Use the syntax:

```
single-valued-path-expression IS [NOT] NULL
```

Example: NULL predicate

```
e.name IS NULL
e.dept.name IS NOT NULL
e.dept IS NOT NULL
```

EMPTY collection predicate:

You can use the EMPTY collection predicate to test if a multivalued relationship has no members.

Use the following syntax:

```
collection-valued-path-expression  IS [NOT]  EMPTY
```

Example: Empty collection predicate

To find all departments with no employees:

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d.emps IS EMPTY
```

MEMBER OF predicate:

This expression tests whether the object reference specified by the single valued path expression or input parameter is a member of the designated collection.

If the collection valued path expression designates an empty collection the value of the MEMBER OF expression is FALSE.

```
{ single-valued-path-expression | input_parameter } [ NOT ] MEMBER [ OF ] collection-valued-path-expression
```

Example: MEMBER OF predicate

Find employees that are not members of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e , DeptBean d
WHERE e NOT MEMBER OF d.emps AND d.deptno = ?1
```

Find employees whose manager is a member of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e, DeptBean d
WHERE e.dept.mgr MEMBER  OF d.emps  and d.deptno=?1
```

EXISTS predicate:

The exists predicate tests for the presence or absence of a condition specified by a subselect.

Use the following syntax:

```
EXISTS ( subselect )
EXISTS collection-valued-path-expression
```

The result of EXISTS is true if the subselect returns at least one value or the path expression evaluates to a nonempty collection, otherwise the result is false.

To negate an EXISTS predicate, precede it with the logical operator NOT.

Example: EXISTS predicate

Return departments that have at least one employee earning more than 1000000:

```
SELECT  OBJECT(d) FROM  DeptBean d
WHERE EXISTS ( SELECT  1  FROM IN (d.emps) e WHERE  e.salary > 1000000 )
```

Return departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d
WHERE NOT EXISTS  ( SELECT 1 FROM IN (d.emps) e)
```

The previous query can also be written as follows:

```
SELECT OBJECT(d) FROM DeptBean d WHERE NOT EXISTS d.emps
```

IS OF TYPE predicate:

The IS OF TYPE predicate is used to test the type of an Enterprise JavaBeans (EJB) reference. It is similar in function to the Java instance of operator.

IS OF TYPE is used when several abstract beans have been grouped into an EJB inheritance hierarchy. The type names specified in the predicate are the bean abstract names. The ONLY option can be used to specify that the reference must be exactly this type and not a subtype.

```
identification-variable IS OF TYPE ( [ONLY] type-1, [ONLY] type-2, ..... )
```

Example: IS OF TYPE predicate

Suppose that bean `ManagerBean` is defined as a subtype of `EmpBean` and `ExecutiveBean` is a subtype of `ManagerBean` in an EJB inheritance hierarchy.

The following query returns employees as well as managers and executives:

```
SELECT  OBJECT(e) FROM EmpBean e
```

If you are interested in objects which are employees and not managers and not executives:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e IS OF TYPE( ONLY EmpBean )
```

If you are interested in object which are managers or executives:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e IS OF TYPE( ManagerBean)
```

The previous query is equivalent to the following query:

```
SELECT  OBJECT(e) FROM ManagerBean e
```

If you are interested in managers only and not executives:

```
SELECT  OBJECT(e) FROM  EmpBean e WHERE e IS OF TYPE( ONLY ManagerBean)
```

or:

```
SELECT  OBJECT(e) FROM  ManagerBean e
WHERE  e IS OF TYPE (ONLY ManagerBean)
```

Scalar functions

An Enterprise JavaBeans (EJB) query contains scalar functions for doing type conversions, string manipulation, and for manipulating date-time values.

The list of scalar functions is documented in the topic EJB query: Scalar functions.

Example: Scalar functions

Find employees hired in 1999:

```
SELECT OBJECT(e) FROM EmpBean e where YEAR(e.hireDate) = 1999
```

The only scalar functions that are guaranteed to be portable across backend datastore vendors are the following:

- ABS
- MOD
- SQRT
- CONCAT
- LENGTH
- LOCATE
- SUBSTRING
- UCASE
- LCASE

The other scalar functions should be used only when DB2 is the backend datastore.

EJB query: Scalar functions:

Enterprise JavaBeans (EJB) query contains scalar built-in functions for doing type conversions, string manipulation, and for manipulating date-time values.

Details of EJB query scalar built-in functions follow:

Numeric functions

```
ABS ( < any numeric datatype > ) -> < any numeric datatype >
MOD ( <int>, <int> ) -> int
SQRT ( < any numeric datatype > ) -> Double
```

Type conversion functions

```
CHAR ( < any  numeric datatype > ) ->  string
CHAR ( <  string  > ) ->  string
CHAR ( < any datetime  datatype >  [, Keyword k ]) ->  string
```

Datetime datatype is converted to its string representation in a format specified by the keyword k. The valid keywords values are ISO, USA, EUR or JIS. If k is not specified the default is ISO.

```
BIGINT ( < any numeric datatype > ) -> Long
BIGINT ( < string > ) -> Long
```

The function in the second line of the following code converts the argument to an integer n by truncation, and returns the date that is n-1 days after January 1, 0001:

```
DATE ( < date string > ) -> Date
DATE (  < any numeric datatype>) -> Date
```

The following function returns date portion of a timestamp:

```
DATE( timestamp ) -> Date
DATE ( < timestamp-string > ) -> Date
```

The following function converts number to decimal with optional precision p and scale s.

```
DECIMAL ( < any numeric datatype > [, p [ ,s ] ] ) -> Decimal
```

The following function converts string to decimal with optional precision p and scale s.

```
DECIMAL ( < string > [ , p [ , s ] ] ) -> Decimal
```

```
DOUBLE ( < any numeric datatype > ) -> Double
DOUBLE ( < string > ) -> Double
```

```
FLOAT ( < any numeric datatype > ) -> Double
FLOAT ( < string > ) -> Double
```

Float is a synonym for DOUBLE.

```
INTEGER ( < any numeric datatype > ) -> Integer
INTEGER ( < string > ) -> Integer
```

```
REAL ( < any numeric datatype > ) -> Float
```

```
SMALLINT ( < any numeric datatype ) -> Short
SMALLINT ( < string > ) -> Short
```

```
TIME ( < time > ) -> Time
TIME ( < time-string  > ) -> Time
TIME ( < timestamp > ) -> Time
TIME ( < timestamp-string  > ) -> Time
```

```
TIMESTAMP ( < timestamp > ) -> Timestamp
TIMESTAMP ( < timestamp-string > ) -> Timestamp
```

String functions

```
CONCAT ( <string>, <string>  ) -> String
```

The following function returns a character string representing absolute value of the argument not including its sign or decimal point. For example, `digits( -42.35)` is “4235”.

```
DIGITS ( Decimal d  ) -> String
```

The following function returns the length of the argument in bytes. If the argument is a numeric or datetime type, it returns the length of internal representation.

```
LENGTH ( < string >  ) -> Integer
```

The following function returns a copy of the argument string where all upper case characters have been converted to lower case.

```
LCASE ( < string > ) -> String
```

The following function returns the starting position of the first occurrence of argument 1 inside argument 2 with optional start position. If not found, it returns 0.

```
LOCATE ( String s1 , String s2  [, Integer start ] ) -> Integer
```

The following function returns a substring of s beginning at character m and containing n characters. If n is omitted, the substring contains the remainder of string s. The result string is padded with blanks if needed to make a string of length n.

```
SUBSTRING ( String s ,  Integer m [ , Integer n ] ) -> String
```

The following function returns a copy of the argument string where all lower case characters have been converted to upper case.

```
UCASE ( < string > ) -> String
```

Date - time functions

The following function returns the day portion of its argument. For a duration, the return value can be -99 to 99.

```
DAY (  Date ) ->  Integer
DAY ( < date-string > ) ->  Integer
DAY ( < date-duration > ) -> Integer
DAY ( Timestamp  ) ->  Integer
DAY ( < timestamp-string > ) ->  Integer
DAY ( < timestamp-duration > ) -> Integer
```

The following function returns one more than number of days from January 1, 0001 to its argument.

```
DAYS ( Date  ) ->  Integer
DAYS ( < Date-string > ) ->  Integer
DAYS ( Timestamp  ) ->  Integer
DAYS ( < timestamp-string > ) ->  Integer
```

The following function returns the hour part of its argument. For a duration, the return value can be -99 to 99.

```
HOUR ( Time ) -> Integer
HOUR ( < time-string > ) -> Integer
HOUR ( < time-duration > ) ->  Integer
HOUR ( Timestamp ) -> Integer
HOUR ( < timestamp-string > ) -> Integer
HOUR ( < timestamp-duration >  ) -> Integer
```

The following function returns the microsecond part of its argument.

```
MICROSECOND ( Timestamp ) -> Integer
MICROSECOND ( < timestamp-string > ) -> Integer
MICROSECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the minute part of its argument. For a duration, the return value can be -99 to 99.

```
MINUTE ( Time ) -> Integer
MINUTE ( < time-string > ) -> Integer
MINUTE ( < time-duration > ) ->  Integer
MINUTE ( Timestamp ) -> Integer
MINUTE ( < timestamp-string > ) -> Integer
MINUTE ( < timestamp-duration >  ) -> Integer
```

The following function returns the month portion of its argument. For a duration, the return value can be -99 to 99.

```
MONTH (  Date ) ->  Integer
MONTH ( < date-string > ) ->  Integer
MONTH ( < date-duration > ) -> Integer
MONTH ( Timestamp  ) ->  Integer
MONTH ( < timestamp-string > ) ->  Integer
MONTH ( < timestamp-duration > ) -> Integer
```

The following function returns the second part of its argument. For a duration, the return value can be -99 to 99.

```
SECOND ( Time ) -> Integer
SECOND ( < time-string > ) -> Integer
SECOND ( < time-duration > ) ->  Integer
SECOND ( Timestamp ) -> Integer
SECOND ( < timestamp-string > ) -> Integer
SECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the year portion of its argument. For a duration, the return value can be -9999 to 9999.

```
YEAR (  Date ) ->  Integer
YEAR ( < date-string > ) ->  Integer
YEAR ( < date-duration > ) -> Integer
YEAR ( Timestamp  ) ->  Integer
YEAR ( < timestamp-string > ) ->  Integer
YEAR ( < timestamp-duration > ) -> Integer
```

Aggregation functions

Aggregation functions operate on a set of values to return a single scalar value. You can use these functions in the select and subselect methods.

The following example illustrates an aggregation:

```
SELECT  SUM (e.salary) FROM EmpBean e WHERE e.dept.deptno =20
```

This aggregation computes the total salary for department 20.

The aggregation functions are AVG, COUNT, MAX, MIN, and SUM. The syntax of an aggregation function is illustrated in the following example:

```
aggregation-function  (    [ ALL |  DISTINCT ]  expression )
```

or:

```
COUNT( [ ALL |  DISTINCT ] identification-variable )
```

or:

```
COUNT( * )
```

The DISTINCT option eliminates duplicate values before applying the function. ALL is the default option and does not eliminate duplicates. Null values are ignored in computing the aggregate function except in the cases of COUNT(*) and COUNT(identification-variable), which return a count of all the elements in the set.

If your datastore is Informix, you must limit the expression argument to a single valued path expression when using the COUNT function or the DISTINCT forms of the functions SUM, AVG, MIN, and MAX.

Defining return type

For a select method using an aggregation function, you can define the return type as a primitive type or a wrapper type. The return type must be compatible with the return type from the datastore. The MAX and MIN functions can apply to any numeric, string or datetime datatype and return the corresponding datatype. The SUM and AVG functions take a numeric type as input, and return the same numeric type that is used in the datastore. The COUNT function can take any datatype, and returns an integer.

When applied to an empty set, the SUM, AVG, MAX, and MIN functions can return a null value. The COUNT function returns zero (0) when it is applied to an empty set. Use wrapper types if the return value might be NULL; otherwise, the container displays an ObjectNotFound exception.

Using GROUP BY and HAVING

The set of values that is used for the aggregate function is determined by the collection that results from the FROM and WHERE clause of the query. You can divide the set into groups and apply the aggregation function to each group. To perform this action, use a GROUP BY clause in the query. The GROUP BY clause defines grouping members, which comprise a list of path expressions. Each path expression specifies a field that is a primitive type of byte, short, int, long, float, double, boolean, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time or java.sql.Timestamp.

The following example illustrates the use of the GROUP BY clause in a query that computes the average salary for each department:

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

In division of a set into groups, a NULL value is considered equal to another NULL value.

Just as the WHERE clause filters tuples (that is, records of the return collection values) from the FROM clause, the groups can be filtered using a HAVING clause that tests group properties involving aggregate functions or grouping members:

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e
GROUP BY e.dept.deptno
HAVING  COUNT(*) > 3  AND  e.dept.deptno > 5
```

This query returns the average salary for departments that have more than three employees and the department number is greater than five.

It is possible to use a HAVING clause without a GROUP BY clause, in which case the entire set is treated as a single group, to which the HAVING clause is applied.

SELECT clause

The SELECT clause consists of either a single identification variable that is defined in the FROM clause, or a single valued path expression that evaluates to an object reference or container managed persistence (CMP) value. You can use the DISTINCT keyword to eliminate duplicate references.

For finder and select queries, the syntax of the SELECT clause is illustrated in the following example:

```
SELECT  [  ALL | DISTINCT  ]
 { single-valued-path-expression  |  aggregation expression  |  OBJECT ( identification-variable )  }
```

For a query that defines a finder method, the query must return an object type consistent with the home that is associated with the finder method. For example, a finder method for a department home can not return employee objects.

Example: SELECT clause

Find all employees that earn more than John:

```
SELECT OBJECT(e) FROM EmpBean ej, EmpBean e
WHERE  ej.name = 'John'  and e.salary > ej.salary
```

Find all departments that have one or more employees who earn less than 20000:

```
SELECT DISTINCT  e.dept  FROM EmpBean e where e.salary < 20000
```

A select method query can have a path expression that evaluates to an arbitrary value:

```
SELECT  e.dept.name  FROM EmpBean e where e.salary < 2000
```

The previous query returns a collection of name values for those departments having employees earning less than 20000.

A select method query can return an aggregate value:

```
SELECT  avg(e.salary)  FROM EmpBean e
```

Example: Valid dynamic queries

For dynamic queries the syntax is as follows:

```
SELECT { ALL | DISTINCT }  [ selection , ]*  selection
selection  ::= { expression   |  scalar-subselect [[AS] id ] }
```

A scalar-subselect is a subselect that returns a single value.

The following are examples of dynamic queries:

```
SELECT e.name, e.salary+e.bonus as total_pay from EmpBean e
SELECT  SUM( e.salary+e.bonus) from EmpBean e where e.dept.deptno = ?1
```

ORDER BY clause

The ORDER BY clause specifies an ordering of the objects in the result collection

Use the syntax:

```
ORDER BY  [ order_element ,]* order_element
order_element ::= { path-expression | integer } [ ASC | DESC ]
```

The path expression must specify a single valued field that is a primitive type of byte, short, int, long, float, double, char or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Character, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp.

ASC specifies ascending order and is the default. DESC specifies descending order.

Integer refers to a selection expression in the SELECT clause.

Example: ORDER BY clause

Return department objects in decreasing deptno order:

```
SELECT  OBJECT(d) FROM  DeptBean d ORDER BY  d.deptno DESC
```

Return employee objects sorted by department number and name:

```
SELECT  OBJECT(e) FROM EmpBean e ORDER BY e.dept.deptno ASC,  e.name DESC
```

UNION clause operation

The UNION clause specifies a combination of the output of two subqueries. The two queries must return the same number of elements and compatible types.

For the purposes of UNION, all Enterprise JavaBeans (EJB) types in the same inheritance hierarchy are considered compatible. UNION requires that equality be defined for the element types.

```
query_expression := query_term [UNION [ALL] query_term]*

query_term :=  {select_clause_dynamic  from_clause  [where_clause]
 [group_by_clause] [having_clause] } | (query_expression) }
```

You cannot use dependent value objects with UNION.

UNION ALL combines all results together in a single collection.

UNION combines results but eliminates duplicates.

If ORDER BY is used together with UNION, the ORDER BY must refer to selection expression using integer numbers.

Examples: UNION operation

This example returns a collection of all employee objects of type EmpBean and all manager objects of type ManagerBean where ManagerBean is a subtype of EmpBean.

```
select e from EmpBean e union all select m from DeptBean d, in(d.mgr) m
```

This example shows a query that is not valid, because EmpBean and DeptBean are not compatible.

```
select e from EmpBean e union all select d from DeptBean d
```

Subqueries

A subquery can be used in quantified predicates, the EXISTS predicate, or the IN predicate. A subquery should only specify a single element in the SELECT clause.

When a path expression appears in a subquery, the identification variable of the path expression must be defined either in the subquery, in one of the containing subqueries, or in the outer query. A scalar subquery is a subquery that returns one value. A scalar subquery can be used in a basic predicate and in the SELECT clause of a dynamic query.

Example: Subqueries

```
SELECT  OBJECT(e) FROM  EmpBean e
WHERE e.salary > ( SELECT  AVG(e1.salary) FROM  EmpBean e1)
```

The previous query returns employees who earn more than average salary of all employees.

```
SELECT  OBJECT(e) FROM EmpBean e WHERE  e.salary >
( SELECT AVG(e1.salary) FROM  IN  (e.dept.emps) e1  )
```

The previous query returns employees who earn more than average salary of their department.

```
SELECT  OBJECT(e) FROM EmpBean e WHERE e.salary =
( SELECT MAX(e1.salary) FROM IN (e.dept.emps) e1  )
```

The previous query returns employees who earn the most in their department.

```
 SELECT OBJECT(e) FROM EmpBean e
WHERE e.salary > ( SELECT AVG(e.salary) FROM EmpBean e1
WHERE YEAR(e1.hireDate) =  YEAR(e.hireDate)  )
```

The previous query returns employees who earn more than the average of employees hired in same year.

EJB query language limitations and restrictions

When using the Enterprise JavaBeans (EJB) query language on the product, deviations can be seen in comparison to standard EJB query language. The limitations and restrictions you must be aware of are listed in the following section.

This topic outlines current known limitations and restrictions.

- EJB query language (QL) queries involving enterprise beans with keys made up of relationships to other enterprise beans appear as not valid and cause errors at deployment time. This is a known problem.
- The IBM EJB QL support extends the EJB 2.0 specification in various ways, including relaxing some restrictions, adding support for more DB2 functions, and so on. If portability across various vendor databases or EJB deployment tools is a concern, then care should be taken to write all EJB QL queries strictly according to Chapter 11 in the EJB 2.0 specification.
- Pre-loading across m:n relationships results in the generation of inaccurate structured query language (SQL). This is a known limitation that may be addressed in the future.
- Pre-loading across self referencing relationships causes inaccurate SQL to be generated.

- Avoid relationships between parent and children enterprise beans within the same inheritance hierarchy that are not well-defined.
- EJB Query Language validation for EJB 2.0 JAR files currently runs as a part of the EJB-RDB Mapping validation. If a mapping document (Map.mapxmi file) does not exist in the project, the EJB queries are not validated.

EJB query compatibility issues with SQL

Because an Enterprise JavaBeans (EJB) query is compiled into structured query language (SQL), you must be aware of compatibility issues between the Java language and SQL.

The two languages differ along the following points that can be critical to correct EJB query formulation:

- The comparison semantics of SQL strings do not exactly match those of the Java language. For example: "A" (the letter A) and"A" (the letter A plus a blank space) are considered equal in SQL, but not in the Java language.
- Comparisons and collating order depend on the underlying database. For example, if you are using DB2 with an EBCDIC code page, the collating order is not the same as doing the sort in a Java program. Some databases sort the NULL value low while others sort the NULL value high.
- An arithmetic overflow causes an exception in SQL, but not in the Java language.
- SQL databases have differing minimum and maximum ranges for floating point values, which can differ from floating point value ranges in the Java language. Values near the range limits of Java Double may fail to translate into SQL.
- Java methods do not translate into SQL; therefore standard EJB queries cannot include Java methods.

  Note: Only with the dynamic EJB query service can you use functions that do not translate into SQL. Such functions include Java methods and converters or composers that are used in mapping enterprise beans to relational databases (RDBs). A standard finder or select query that uses any of these functions fails at deployment time with the message "Cannot push down query". (You can resolve this problem by changing either the query or the mapping.) The dynamic query run time, however, processes the query by performing the operation involving the function in the application server.

Database restrictions for EJB query

The Enterprise JavaBeans (EJB) query functions must adhere to certain restrictions for databases.

General database restriction

- All of the enterprise beans involved in a given query must map to the same data source. The EJB query does not support cross-data source join operations.
- It is possible that a structured query language (SQL) statement generated by the WebSphere Application Server deployment code generation utility for an *ejbSelect* EJB query language query returns rows in a result set that consist of null values in all columns.

  During run time persistence manager saves the set received as a result from this query. When your application retrieves the primary key of the result bean, persistence manager calls the extractor. The extractor is a method that is an EJB deploy generated class. This method returns a value of 0 for any null column entries. This value is passed back to the EJB container to forward to the application. The EJB container invokes the bean instance with the PK value of 0. This could create a problem, as the end user cannot determine if this bean instance has a *null PK* or a *PK value of 0*.

  To avoid this, use the *IS NOT NULL* clause in the finder query to eliminate such null values from the result set.

Specific database restrictions

Different database products place different restrictions on elements that can be included in EJB query statements. Following is a list of those restrictions; check with your database administrator to see if any apply in your environment:

- Certain functions are used in queries that run against DB2 only, because these functions are not supported by other databases. These functions include date and time arithmetic expressions, certain scalar functions including those not listed as portable across vendors, and implied scalar functions when used for mapping certain container managed persistence (CMP) fields. For example, consider mapping an int numeric type to a decimal (5,2) type field. When deployed against a database other than DB2, a finder or select query that contains a CMP field with this particular mapping fails, producing a `Cannot push down query` error message.
- A CMP of type String, when mapped to a character large object (CLOB) in the database, cannot be used in comparison operations because the database does not support CLOB comparisons.
- Databases can impose limits on the length of string values that are used either as literals or input parameters with comparison operators. These limits can hinder query performance. For example: For DB2 on the z/OS platform, the search “name = ?1” can fail if the value of ?1 at run time is greater than 255 in length.
- Mapping a numeric CMP type to a column that contains a dissimilar type can cause unexpected results. For example, consider the case of mapping the int numeric type to a column of type decimal (5,2). This scenario does not preserve an exact decimal value (for example, the value 12.25) over the course of transfer from the database to the enterprise bean CMP field, and back again to the database. This mapping causes replacement of the initial value with a whole number (in this case, 12). Consequently, you want to avoid using the CMP field in comparison operations when the CMP field uses a mapping of this nature.
- Some databases do not support a data type that corresponds to the semantics of java.sql.Time. For example: If a CMP field of type java.sql.Time is mapped to an Oracle DATE column, comparisons on time might not produce the expected result because the year-month-day portion of the column value is truncated in the mapping.
- Some databases treat a zero length string value ('') as a null value; this approach can affect the query results. For the sake of portability, avoid the use of zero length string values.
- Some databases perform division between two integer values using integer arithmetic rules, while others use non-integer rules. This discrepancy might not be desirable in environments that use both kinds of databases. For the sake of portability, avoid the division of integer values in an EJB query.

Rules for data type manipulation in EJB query

When using an Enterprise JavaBeans (EJB) query to work with data types, certain rules must be followed.

You can use a CMP field of any type in a SELECT clause. You must, however, use fields of only the following types in search conditions and in grouping or ordering operations:

- Primitive types: byte, short, int, long, float, double, boolean, char
- Object types: Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.util.Date
- JDBC types: java.sql.Date, java.sql.Time, java.sql.Timestamp
- Binary string: byte

If ALL of the following conditions occur:
- a CMP field of one of the basic types listed previously is mapped to an SQL column using a converter
- the CMP field appears prior to a basic predicate
- following the predicate is a literal or input parameter

then the toData() method of the converter is used to compute the SQL search value.

For example, given a converter that maps the integer value 10 to the string value “Ten” the following EJB query:

```
e.cmp = 10
```

is translated into the following SQL query:

```
column = 'Ten'
```

If you include a more complicated predicate, such as in the following example:

```
e.cmp * 10 >  e.salary
```

in a finder or select query, you receive the `Cannot push down query` error message. Use the dynamic EJB query service for such multi-function queries; the dynamic query run time processes the predicate in the application server.

Overall, converters preserve equality, collating sequence, and NULL values. If a converter does not meet these requirements, avoid using it for CMP field comparison operations.

A user type cannot be used in a comparison operation or expression. You can, however, use subfields of the user type in a path expression. For example, consider the CMP addr field with the type `com.exam.Address`, and street, city, and state subfields. The following syntax for a query on this CMP field is not valid:

```
e.addr =  ?1
```

However, a query that designates one of the subfields is valid:

```
e.addr.street = ?1
```

A CMP field can be mapped to an SQL column using Java serialization. Using the CMP field in predicates or expressions for deployment queries usually results in the `Cannot push down query` error message. The dynamic query run time processes the expression by reading and deserializing all instances of the user type in the application server.

However, this expensive process sacrifices performance. You can maintain performance by using a composer in a deployment EJB query. In the previous example, if you want to map the addr field to a binary type, you use a composer to map each subfield to a binary column in the database.

EJB query: Reserved words

The following words are reserved in WebSphere Application Server Enterprise JavaBeans (EJB) queries.

all, as, distinct, empty, false, from, group, having, in, is, like, select, true, union, where

Avoid using identifiers that start with underscore (for example, _integer) as these are also reserved.

EJB query: BNF syntax

The Backus-Naur Form (BNF) is one of the most commonly used notations for specifying the syntax of programming languages or command sets. This article lists the syntax for Enterprise JavaBeans (EJB) query language.

```
EJB QL ::= [select_clause] from_clause [where_clause] [order_by_clause]

DYNAMIC EJB QL := query_expression  [order_by_clause]

query_expression := query_term [UNION [ALL] query_term]*

query_term :=  {select_clause_dynamic  from_clause  [where_clause]
 [group_by_clause] [having_clause] } | (query_expression) }  [order_by_clause]


from_clause::=FROM identification_variable_declaration
               [, {identification_variable_declaration | collection_member_declaration } ]*

identification_variable_declaration::=collection_member_declaration |
       range_variable_declaration [join]*

join :=  [ { LEFT [OUTER] | INNER  }]  JOIN  {collection_valued_path_expression | single_valued_path_expression}
 [AS] identifier

collection_member_declaration::=
       IN ( collection_valued_path_expression ) [AS] identifier
```

```
range_variable_declaration::=abstract_schema_name [AS] identifier

single_valued_path_expression ::=
       {single_valued_navigation | identification_variable}. ( cmp_field |
 method  |  cmp_field.value_object_attribute |  cmp_field.value_object_method )
          | single_valued_navigation

single_valued_navigation::=
       identification_variable.[  single_valued_cmr_field.  ]*
            single_valued_cmr_field

collection_valued_path_expression ::=
       identification_variable.[  single_valued_cmr_field.  ]*
           collection_valued_cmr_field

select_clause::= SELECT { ALL | DISTINCT }  {single_valued_path_expression |
                         identification_variable |  OBJECT ( identification_variable) |
        aggregate_functions }

select_clause_dynamic ::= SELECT { ALL | DISTINCT }  [ selection , ]*  selection

selection  ::= { expression  |  subselect }  [[AS] id ]

order_by_clause::= ORDER BY [ {single_valued_path_expression | integer}  [ASC|DESC],]*
      {single_valued_path_expression | integer}[ASC|DESC]

where_clause::= WHERE conditional_expression

conditional_expression ::= conditional_term |
                           conditional_expression OR conditional_term
conditional_term ::= conditional_factor |
                     conditional_term AND conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary::=simple_cond_expression | (conditional_expression)

simple_cond_expression ::= comparison_expression | between_expression |
       like_expression | in_expression | null_comparison_expression |
       empty_collection_comparison_expression | quantified_expression |
       exists_expression | is_of_type_expression  |  collection_member_expression

between_expression ::= expression [NOT] BETWEEN expression AND expression

in_expression ::= single_valued_path_expression [NOT] IN
          { (subselect) |  ( [ atom ,]* atom )  }

atom = { string-literal | numeric-constant | input-parameter }

like_expression ::= expression [NOT] LIKE
                {string_literal | input_parameter}
                [ESCAPE {string_literal | input_parameter}]

null_comparison_expression ::=
       single_valued_path_expression IS [ NOT ] NULL

empty_collection_comparison_expression ::=
       collection_valued_path_expression IS [NOT] EMPTY

collection_member_expression  ::=
       { single_valued_path_expression |  input_paramter }  [ NOT ] MEMBER [ OF ]
         collection_valued_path_expression

quantified_expression ::=
       expression comparison_operator  {SOME | ANY | ALL} (subselect)

exists_expression ::= EXISTS {collection_valued_path_expression |  (subselect)}

subselect ::= SELECT [{ ALL | DISTINCT }]  expression  from_clause [where_clause]
       [group_by_clause] [having_clause]

group_by_clause::= GROUP BY [single_valued_path_expression,]*
                    single_valued_path_expression
```

```
having_clause ::= HAVING conditional_expression


is_of_type_expression ::= identifier  IS OF TYPE
            ([[ONLY] abstract_schema_name,]* [ONLY] abstract_schema_name)

comparison_expression ::=  expression   comparison_operator { expression |  ( subquery ) }

comparison_operator ::=    = | > | >= | < | <= | <>

method ::=  method_name( [[expression , ]* expression ] )

expression ::= term |   expression {+|-} term

term ::=  factor |  term {*|/} factor

factor ::= {+|-} primary

primary ::= single_valued_path_expression | literal |
       ( expression ) |  input_parameter | functions | aggregate_functions

aggregate_functions :=
       AVG([ALL|DISTINCT] expression) |
       COUNT({[ALL|DISTINCT] expression | * | identification_variable }) |
       MAX([ALL|DISTINCT] expression) |
       MIN([ALL|DISTINCT] expression) |
       SUM([ALL|DISTINCT] expression) |

functions ::=
        ABS(expression) |
        BIGINT(expression) |
        CHAR({expression [,{ISO|USA|EUR|JIS}] )  |
        CONCAT (expression , expression ) |
        DATE(expression) |
        DAY({expression ) |
        DAYS( expression) |
        DECIMAL( expression [,integer[,integer]])
        DIGITS( expression) |
        DOUBLE( expression ) |
        FLOAT( expression) |
        HOUR ( expression ) |
        INTEGER( expression ) |
        LCASE ( expression) |
        LENGTH(expression) |
        LOCATE( expression, expression [, expression] ) |
        MICROSECOND( expression ) |
        MINUTE ( expression ) |
        MOD ( expression , expression ) |
        MONTH( expression ) |
        REAL( expression) |
        SECOND( expression ) |
        SMALLINT( expression )  |
        SQRT (  expression) |
        SUBSTRING( expression, expression[, expression]) |
        TIME( expression ) |
        TIMESTAMP( expression ) |
        UCASE ( expression) |
        YEAR( expression )

xrel := XREL identification_variable . { single_valued_cmr_field | collection_valued_cmr_field }
             [ , identification_variable . { single_valued_cmr_field | collection_valued_cmr_field } ]*
```

EJB specification and WebSphere query language comparison

WebSphere Application Server extends the Enterprise JavaBeans (EJB) query language with elements of its own.

WebSphere Application Server supports the following extensions to the EJB query language.

Table 46. Extensions supported for the EJB query language. The product supports the following extensions.

Item	
Delimited identifiers	
Dependent Value object attributes used in path expressions	
EJB Inheritance	
EXISTS predicate	
Java methods: EJB bean methods or value object methods	dynamic query only
Multiple element select clauses	dynamic query only
SQL Date/time expressions	
Subqueries, group by, and having clauses	

Using the dynamic query service

There are times in the development process when you might prefer to use the dynamic query service rather than the regular Enterprise JavaBeans (EJB) query service (which can be referred to as *deployment query*). During testing, for instance, the dynamic query service can be used at application run time, so you do not have to re-deploy your application.

About this task

Following are common reasons for using the dynamic query service rather than the regular EJB query service:

- You need to programmatically define a query at application run time, rather than at deployment.
- You need to return multiple CMP or CMR fields from a query. (Deployment queries allow only a single element to be specified in the SELECT clause.) For more information, see the topic, Example: EJB queries.
- You want to return a computed expression in the query.
- You want to use value object methods or bean methods in the query statement. For more information, see the topic, Path expressions.
- You want to interactively test an EJB query during development, but do not want to repeatedly deploy your application each time you update a finder or select query.

The dynamic query API is a stateless session bean; using it is similar to using any other J2EE EJB application bean. It is included in the com.ibm.websphere.ejbquery in the API package.

The dynamic query bean has both a remote and a local interface. If you want to return remote EJB references from the query, or if the query statement contains remote methods, you must use the query remote interface:

remote interface = `com.ibm.websphere.ejbquery.Query`
remote home interface = `com.ibm.websphere.ejbquery.QueryHome`

If you want to return local EJB references from the query, or if the query statement contains local methods, you must use the query local interface:

local interface = `com.ibm.websphere.ejbquery.QueryLocal`
local home interface = `com.ibm.websphere.ejbquery.QueryLocalHome`

Because it uses less application server memory, the local interface ensures better overall EJB performance than the remote.

Procedure

1. Verify that the query.ear application file is installed on the application server on which your application is to run, if that server is different from the default application server created during installation of the product.

   The `query.ear` file is located in the *app_server_root* directory, where `<WAS_HOME>` is the location of the WebSphere Application Server. The product installation program installs the `query.ear` file on the default application server using a JNDI name of

   ```
   com/ibm/websphere/ejbquery/Query
   ```

   (You or the system administrator can change this name.)
2. Set up authorization for the methods executeQuery(), prepareQuery(), and executePlan() in the remote and local dynamic query interfaces to control access to sensitive data. (This step is necessary only if your application requires security.)

   Because you cannot control which ASN names, CMP fields, or CMR fields can be used in a dynamic EJB query, you or your system administrator must place restrictions on use of the methods. If, for example, a user is permitted to run the executeQuery method, he or she can run any valid dynamic query. In a production environment, you certainly want to restrict access to the remote query interface methods.
3. Write the dynamic query as part of your application client code. You can refer to the example topics, Remote interface dynamic query example, and Local interface dynamic query example, as query models; they illustrate which import statements to use.
4. If the CMP you want to query is on a different module, you should:
   a. do a remote lookup on query.ear
   b. map the query.ear file to the server that the queried CMP bean is installed on.
5. Compile and run your client program with the file **qryclient.jar** in the classpath.

Example

Using the remote interface for Dynamic query.

When you run a dynamic Enterprise JavaBeans (EJB) query using the remote interface, you are calling the executeQuery method on the Query interface. The executeQuery method has a transaction attribute of REQUIRED for this interface; therefore you do not need to explicitly establish a transaction context for the query to run.

When you run a dynamic Enterprise JavaBeans (EJB) query using the remote interface, you are calling the executeQuery method on the Query interface. The executeQuery method has a transaction attribute of REQUIRED for this interface; therefore you do not need to explicitly establish a transaction context for the query to run.

Begin with the following import statements:

```
import com.ibm.websphere.ejbquery.QueryHome;
import com.ibm.websphere.ejbquery.Query;
import com.ibm.websphere.ejbquery.QueryIterator;
import com.ibm.websphere.ejbquery.IQueryTuple;
import com.ibm.websphere.ejbquery.QueryException;
```

Next, write your query statement in the form of a string, as in the following example that retrieves the names and ejb-references for underpaid employees:

```
String query =
"select e.name as name , object(e) as emp from EmpBean e where e.salary < 50000";
```

Create a Query object by obtaining a reference from the QueryHome class. (This class defines the executeQuery method.) Note that for the sake of simplicity, the following example uses the dynamic query JNDI name for the Query object:

```
InitialContext ic =  new InitialContext();

Object obj =  ic.lookup("com/ibm/websphere/ejbquery/Query");

QueryHome  qh =
 ( QueryHome) javax.rmi.PortableRemoteObject.narrow( obj, QueryHome.class );
Query qb = qh.create();
```

You then must specify a maximum size for the query result set, which is defined in the QueryIterator object, which is included in the Class QueryIterator. This class is included in the You then must specify a maximum size for the query result set, which is defined in the QueryIterator object, which is included in the QueryIterator API package. This example sets the maximum size of the result set to 99:

```
QueryIterator it = qb.executeQuery(query, null, null ,0, 99 );
```

The iterator contains a collection of IQueryTuple objects, which are records of the return collection values. Corresponding to the criteria of our example query statement, each tuple in this scenario contains one value of *name* and one value of *object(e)*. To display the contents of this query result, use the following code:

```
while (it.hasNext() ) {
 IQueryTuple tuple = (IQueryTuple) it.next();
 System.out.print( it.getFieldName(1) );
 String s = (String) tuple.getObject(1);
 System.out.println( s);
 System.out.println( it.getFieldName(2) );
 Emp e = ( Emp) javax.rmi.PortableRemoteObject.narrow( tuple.getObject(2), Emp.class );
 System.out.println( e.getPrimaryKey().toString());
}
```

The output from the program might look something like the following:

```
name Bob
emp 1001
name Dave
emp 298003
...
```

Finally, catch and process any exceptions. An exception might occur because of a syntax error in the query statement or a run-time processing error. The following example catches and processes these exceptions:

```
} catch (QueryException qe) {
    System.out.println("Query Exception "+ qe.getMessage() );
}
```

Handling large result collections for the remote interface query

If you intend your query to return a large collection, you have the option of programming it to return results in multiple smaller, more manageable quantities. Use the skipRow and maxRow parameters on the remote executeQuery method to retrieve the answer in chunks. For example:

```
int skipRow=0;
int maxRow=100;
QueryIterator it = null;
do {
 it = qb.executeQuery(query, null, null ,skipRow, maxRow );
 while (it.hasNext() ) {
 // display result
 skipRow = skipRow + maxRow;
}
} while ( ! it.isComplete() ) ;
```

Using the local interface for Dynamic query.

When you run a dynamic Enterprise JavaBeans (EJB) query using the local interface, you are calling the executeQuery method on the QueryLocal interface. This interface does not initiate a transaction for the method; therefore you must explicitly establish a transaction context for the query to run.

Note: To establish a transaction context, the following example calls the begin() and commit() methods. An alternative to using these methods is simply embedding your query code within an EJB method that runs within a transaction context.

Begin your query code with the following import statements:

```
import com.ibm.websphere.ejbquery.QueryLocalHome;
import com.ibm.websphere.ejbquery.QueryLocal;
import com.ibm.websphere.ejbquery.QueryLocalIterator;
import com.ibm.websphere.ejbquery.IQueryTuple;
import com.ibm.websphere.ejbquery.QueryException;
```

Next, write your query statement in the form of a string, as in the following example that retrieves the names and ejb-references for underpaid employees:

```
String query =
"select e.name, object(e) from EmpBean e where e.salary < 50000 ";
```

Create a QueryLocal object by obtaining a reference from the QueryLocalHome class. (This class defines the executeQuery method.) Note that in the following example, `ejb/query` is used as a local EJB reference pointing to the dynamic query JNDI name (`com/ibm/websphere/ejbquery/Query`):

```
InitialContext ic =  new InitialContext();
   QueryLocalHome  qh =  ( LocalQueryHome) ic.lookup( "java:comp/env/ejb/query" );
QueryLocal qb = qh.create();
```

The last portion of code initiates a transaction, calls the executeQuery method, and displays the query results. The QueryLocalIterator class is instantiated because it defines the query result set. This class is included in the Class QueryIterator API package. Keep in mind that the iterator loses validity at the end of the transaction; you must use the iterator in the same transaction scope as the executeQuery call.

```
userTransaction.begin();
QueryLocalIterator it = qb.executeQuery(query, null, null);
while (it.hasNext() ) {
 IQueryTuple tuple = (IQueryTuple) it.next();
 System.out.print( it.getFieldName(1) );
 String s = (String) tuple.getObject(1);
 System.out.println( s);
 System.out.println( it.getFieldName(2) );
 EmpLocal e = ( EmpLocal ) tuple.getObject(2);
 System.out.println( e.getPrimaryKey().toString());
}
userTransaction.commit();
```

In most situations, the QueryLocalIterator object is *demand-driven*. That is, it causes data to be returned incrementally: for each record retrieval from the database, the next() method must be called on the iterator. (Situations can exist in which the iterator is not demand-driven. For more information, consult the "Local query interfaces" subsection of the Dynamic query performance considerations topic.)

Because the full query result set materializes incrementally in the application server memory, you can easily control its size. During a test run, for example, you may decide that return of only a few tuples of the query result is necessary. In that case you should use a call of the close() method on the QueryLocalIterator object to close the query loop. Doing so frees SQL resources that the iterator uses. Otherwise, these resources are not freed until the full result set accumulates in memory, or the transaction ends.

Example: Using the remote interface for Dynamic query

When you run a dynamic Enterprise JavaBeans (EJB) query using the remote interface, you are calling the executeQuery method on the Query interface. The executeQuery method has a transaction attribute of REQUIRED for this interface; therefore you do not need to explicitly establish a transaction context for the query to run.

Begin with the following import statements:

```
import com.ibm.websphere.ejbquery.QueryHome;
import com.ibm.websphere.ejbquery.Query;
import com.ibm.websphere.ejbquery.QueryIterator;
import com.ibm.websphere.ejbquery.IQueryTuple;
import com.ibm.websphere.ejbquery.QueryException;
```

Next, write your query statement in the form of a string, as in the following example that retrieves the names and ejb-references for underpaid employees:

```
String query =
"select e.name as name , object(e) as emp from EmpBean e where e.salary < 50000";
```

Create a Query object by obtaining a reference from the QueryHome class. (This class defines the executeQuery method.) Note that for the sake of simplicity, the following example uses the dynamic query JNDI name for the Query object:

```
InitialContext ic =  new InitialContext();

Object obj =  ic.lookup("com/ibm/websphere/ejbquery/Query");

QueryHome  qh =
 ( QueryHome) javax.rmi.PortableRemoteObject.narrow( obj, QueryHome.class );
Query qb = qh.create();
```

You then must specify a maximum size for the query result set, which is defined in the QueryIterator object, which is included in the Class QueryIterator. This class is included in the You then must specify a maximum size for the query result set, which is defined in the QueryIterator object, which is included in the QueryIterator API package. This example sets the maximum size of the result set to 99:

```
QueryIterator it = qb.executeQuery(query, null, null ,0, 99 );
```

The iterator contains a collection of IQueryTuple objects, which are records of the return collection values. Corresponding to the criteria of our example query statement, each tuple in this scenario contains one value of *name* and one value of *object(e)*. To display the contents of this query result, use the following code:

```
while (it.hasNext() ) {
 IQueryTuple tuple = (IQueryTuple) it.next();
 System.out.print( it.getFieldName(1) );
 String s = (String) tuple.getObject(1);
 System.out.println( s);
 System.out.println( it.getFieldName(2) );
 Emp e = ( Emp) javax.rmi.PortableRemoteObject.narrow( tuple.getObject(2), Emp.class );
 System.out.println( e.getPrimaryKey().toString());
}
```

The output from the program might look something like the following:

```
name Bob
emp 1001
name Dave
emp 298003
...
```

Finally, catch and process any exceptions. An exception might occur because of a syntax error in the query statement or a run-time processing error. The following example catches and processes these exceptions:

```
} catch (QueryException qe) {
    System.out.println("Query Exception "+ qe.getMessage() );
}
```

Handling large result collections for the remote interface query

If you intend your query to return a large collection, you have the option of programming it to return results in multiple smaller, more manageable quantities. Use the skipRow and maxRow parameters on the remote executeQuery method to retrieve the answer in chunks. For example:

```
int skipRow=0;
int maxRow=100;
QueryIterator it = null;
do {
 it = qb.executeQuery(query, null, null ,skipRow, maxRow );
 while (it.hasNext() ) {
 // display result
 skipRow = skipRow + maxRow;
}
} while ( ! it.isComplete() ) ;
```

Example: Using the local interface for Dynamic query

When you run a dynamic Enterprise JavaBeans (EJB) query using the local interface, you are calling the executeQuery method on the QueryLocal interface. This interface does not initiate a transaction for the method; therefore you must explicitly establish a transaction context for the query to run.

Note: To establish a transaction context, the following example calls the begin() and commit() methods. An alternative to using these methods is simply embedding your query code within an EJB method that runs within a transaction context.

Begin your query code with the following import statements:

```
import com.ibm.websphere.ejbquery.QueryLocalHome;
import com.ibm.websphere.ejbquery.QueryLocal;
import com.ibm.websphere.ejbquery.QueryLocalIterator;
import com.ibm.websphere.ejbquery.IQueryTuple;
import com.ibm.websphere.ejbquery.QueryException;
```

Next, write your query statement in the form of a string, as in the following example that retrieves the names and ejb-references for underpaid employees:

```
String query =
"select e.name, object(e) from EmpBean e where e.salary < 50000 ";
```

Create a QueryLocal object by obtaining a reference from the QueryLocalHome class. (This class defines the executeQuery method.) Note that in the following example, `ejb/query` is used as a local EJB reference pointing to the dynamic query JNDI name (`com/ibm/websphere/ejbquery/Query`):

```
InitialContext ic =  new InitialContext();
   QueryLocalHome  qh =  ( LocalQueryHome) ic.lookup( "java:comp/env/ejb/query" );
QueryLocal qb = qh.create();
```

The last portion of code initiates a transaction, calls the executeQuery method, and displays the query results. The QueryLocalIterator class is instantiated because it defines the query result set. This class is included in the Class QueryIterator API package. Keep in mind that the iterator loses validity at the end of the transaction; you must use the iterator in the same transaction scope as the executeQuery call.

```
userTransaction.begin();
QueryLocalIterator it = qb.executeQuery(query, null, null);
while (it.hasNext() ) {
 IQueryTuple tuple = (IQueryTuple) it.next();
 System.out.print( it.getFieldName(1) );
 String s = (String) tuple.getObject(1);
 System.out.println( s);
```

```
 System.out.println( it.getFieldName(2) );
 EmpLocal e = ( EmpLocal ) tuple.getObject(2);
 System.out.println( e.getPrimaryKey().toString());
}
userTransaction.commit();
```

In most situations, the QueryLocalIterator object is *demand-driven*. That is, it causes data to be returned incrementally: for each record retrieval from the database, the next() method must be called on the iterator. (Situations can exist in which the iterator is not demand-driven. For more information, consult the "Local query interfaces" subsection of the Dynamic query performance considerations topic.)

Because the full query result set materializes incrementally in the application server memory, you can easily control its size. During a test run, for example, you may decide that return of only a few tuples of the query result is necessary. In that case you should use a call of the close() method on the QueryLocalIterator object to close the query loop. Doing so frees SQL resources that the iterator uses. Otherwise, these resources are not freed until the full result set accumulates in memory, or the transaction ends.

Dynamic query performance considerations

While using a dynamic query can be convenient, there are times when it can have an impact on your application performance.

General performance considerations

Use of the following elements in your dynamic query can diminish application performance somewhat:

- Datatype converters and Java methods

  Why: In general, query operations and predicates are translated into SQL so that the database server can perform them. If your query includes datatype converters (for EJB to RDB mapping, for example) or Java methods, however, the associated predicates and operations of your query must be performed in the memory of the application server.
- EJB methods and criteria that call for the return of EJB references

  Why: Queries that incorporate these elements trigger full activation of EJBs in the memory of the application server. (Returning a list of CMP fields from a query does not cause an EJB to be activated.)

When assessing application performance, you should also be aware that dynamic queries share connections with the persistence manager. Consequently, an application that includes a mixture of finder methods, CMR navigation, and dynamic queries relies on a single shared connection between the persistence manager and the dynamic query service to perform these tasks.

Limiting the return collection size

- **Remote interface queries**: The QueryIterator class of the remote interface mandates that all of your query results materialize in application server memory over the course of one method call. The SQL cursor(s) used to run the EJB query are closed upon completion of that call. Because this requirement poses a high risk for creating bottlenecks within the database server, you need to limit the size of any potentially large result collections.
- **Local interface queries**: In most situations, the QueryLocalIterator object behaves as a wrapper around an SQL cursor. It is *demand-driven*; it causes data to be returned incrementally. For each record retrieval from the database, the next() method must be called on the iterator.

  Use of certain operations in local interface queries, however, overrides the demand-driven behavior. In these cases, the query results fully materialize in memory just as do the result collections of remote interface queries. An example of such a case is:

  ```
  select  e.myBusinessMethod( ) from EmpBean e
   where e.salary < 50000 order by 1 desc
  ```

  This query requires performance of an EJB method to produce the final result collection. Consequently, the full dataset from the database must be returned in one collection to application server memory,

where the EJB method can be run on the dataset in its entirety. For that reason, local interface query operations that invoke EJB methods are generally not demand-driven. You cannot control the return collection size for such queries.

Because they *are* demand-driven, all other local interface queries allow you to control the size of return collections. You can use a call of the close() method on the QueryLocalIterator object to close the query loop after the desired number of return values has been fetched from the datastore. Otherwise, the SQL cursor(s) used to run the EJB query are not closed until the full result set accumulates in memory, or the transaction ends.

Access intent implications for dynamic query

WebSphere Application Server gives you the option to set access intent policies for your entity enterprise beans as a way of managing their transfer of data with the underlying data store. An access intent policy controls the isolation level used on the data source connection, as well as the database locks used during data retrieval. By manipulating these elements, you can maximize the efficiency of your application's data flow.

To learn more, begin with the topic, Access intent policies and the topic, Concurrency control.

When formulating dynamic queries, keep in mind the following considerations concerning their interaction with access intent policies:

- A dynamic query uses the first ASN name in the FROM clause to determine access intent.
- The collection increment attribute of an access intent policy is not used in processing a dynamic query.
- When performed on entity beans that have a pessimistic-Update access intent policy, your dynamic queries must return updateable collections. Therefore you need to formulate your query statements to return only collections of entity beans, *not* collections of CMP fields. For example, the statement `select object(c) from Customer` is valid for a dynamic query performed under the constraint of a pessimistic-Update policy. The statement `select c.name from Customer c`, however, is not a valid dynamic query under this constraint.
- Using pessimistic-Update policy places restrictions on the types of query expressions. The restrictions depend on the back end database type and release. Refer to the topic Access intent – isolation levels and update locks, for details.

Dynamic query API: prepareQuery() and executePlan() methods

Use these methods to more efficiently allocate the overhead associated with dynamic query. They are equivalent in function to the prepareStatement() and executeQuery() methods of the JDBC API.

To perform a dynamic Enterprise JavaBeans (EJB) query, the application server must parse the query string into structured query language (SQL) at run time. You can, of course, eliminate run-time overhead by choosing to perform a standard EJB query instead of a dynamic query. Sometimes referred to as *deployment queries*, standard queries are parsed and built at deployment, then performed by a finder or select method.

Another option is to write code that redistributes dynamic query overhead for better application performance. Begin by calling the prepareQuery() method in place of the executeQuery() method. The prepareQuery() method parses and translates your query, and returns a string called a *query plan*. The plan contains the SQL statement produced by parsing and translation, as well as other information needed by the dynamic query API. Save this string in your application and call the executePlan() method with the string to run your query. (You also might want to use the prepareQuery() method simply to see the SQL translation product; just call the method and display the return value.)

Pass the parameters of your query as an array of type Object on the prepareQuery() and the executePlan() method calls. Ensure that you pass appropriate data types, because the application server validates your query according to parameter type (rather than actual values) when it processes the prepareQuery() method call.

Example code

Note: In the example code that follows, the first executePlan() method call substitutes `parms[0]` for `?1`. Hence the first query performed is functionally equivalent to the following query statement:

```
select e.name as name, object(e) as emp from EmpBean e where e.salary < 50000
```

The second call runs a query that is functionally equivalent to this statement:

```
select e.name as name, object(e) as emp from EmpBean e where e.salary < 60000
```

The example:

```
String query =
"select e.name as name , object(e) as emp from EmpBean e where e.salary < ?1";
QueryIterator it = null;
Integer[] parms = new Integer[1];
parms[0] = new Integer(0);
```

In the call to `prepareQuery()`, pass any Integer value. Doing so defines `?1` as an Integer type, as in the following:

```
String queryPlan= qb.prepareQuery(query, parms, null );

   parms[0] = new Integer(50000);
```

Next you run the query with a real value of `Integer(50000)` for `?1`:

```
select e.name as name, object(e) as emp from EmpBean e where e.salary < 50000it =
 qb.executePlan( queryPlan, parms, 0, 99);

parms[0] = new Integer(60000);
```

Run the query again with a different value of `Integer(60000)` for `?1`:

```
it = qb.executePlan( queryPlan, parms, 0, 99);
```

Dynamic and deployment EJB query services comparison

You can use the dynamic query service to build and execute queries against entity beans constructed dynamically at run time, rather than defining them at deployment time. By using dynamic query you gain the flexibility of queries defined at run time and utilize the power of Enterprise JavaBeans (EJB)-Query Language (QL). Apart from supporting all of the capabilities of an EJB-QL query, dynamic query adds functionality not available to standard static query. Two examples are the ability to select multiple data fields directly from the bean itself (static queries currently only allow one) and executing business methods directly in the query.

You can effectively create more efficient and less resource intensive applications with dynamic query. For example, two data fields are required from the results of a query. Because a standard EJB-QL query can only select one data field, it is necessary to select the entire EJB object and extract the needed data from the returned results through data access methods, possibly traversing Container Managed Relationships (CMR) boundaries in the process. However, when using dynamic query, you can get both pieces of data directly from the query without additional CMR traversal or accessor methods. This principle is the key to evaluating whether or not dynamic query can be used for performance gain. You should review the amount of data that must be retrieved, in addition to the amount of business logic needed to retrieve it, for example, CMR traversal or accessor methods.

Using parameters in the query rather than literal values is another performance consideration. Under most circumstances, it is better to define conditional values as parameters in the query and then pass those parameters through the appropriate mechanisms. By using this method, you have a greater chance of matching a cached query plan, and you eliminate the need to parse and build the plan from scratch. For example, “SELECT Object(o) FROM schemaname AS o WHERE o.fieldname LIKE foo”, is more appropriately expressed as “SELECT Object(o) FROM schemaname AS o WHERE o.fieldname LIKE ?1” with the value foo passed as a parameter to the executeQuery method. The result is that any subsequent

execution of a dynamic query structure that is the same, except for different string literal conditions, is registered as a plan cache hit (which delivers better “observed” performance).

When used as a direct replacement for an equivalent static query, dynamic query is approximately 25% slower than the static variation. This slowdown is due to the need for parsing and building a plan for the query, in addition to executing it. In the static variation, these costs are paid at deploy time. Despite this, the added functionality gained through the use of dynamic query, specifically the ability to select multiple data fields in a single query even across CMRs, creates opportunities to utilize dynamic query for the sake of performance improvement.

Chapter 10. Developing EJB applications

This page provides a starting point for finding information about enterprise beans.

Based on the Enterprise JavaBeans (EJB) specification, enterprise beans are Java components that typically implement the business logic of Java 2 Platform, Enterprise Edition (J2EE) applications as well as access data.

Developing EJB 2.x enterprise beans

Partial column update feature for container managed persistence

The Container Managed Persistence (CMP) bean method `ejbStore` stored all of the persistent attributes of the CMP bean to the database, even if only a subset of persistent attribute fields were changed. This needless performance degradation is eliminated in this release of the product.

Note: Entity beans are not supported in EJB 3.0 modules.

For Enterprise JavaBeans (EJB) 2.x CMP entity beans, you can use the partial update feature to specify how you want to update the persistent attributes of the CMP bean to the database. This feature is provided as a bean level persistence option, called `PartialOperation`, in the access intent policy configured for the bean. PartialOperation has two possible values:

NONE Partial update is turned off. All of the persistent attributes of the CMP bean are stored to the database. This is the default value.

UPDATE_ONLY
Specifies that updates to the database occur only for the persistent attributes of the CMP bean that are changed.

For information on how to set partial update, see “Setting partial update for container-managed persistent beans” on page 315.

Performance

Performing partial updates increases performance in several ways:

- by reducing query execution time, since only a subset of the columns are in the query. Improvement is increased for tables with many columns and indexes. When the table has many indexes only the indexes affected by the updated columns need to be updated by the backend database.
- by reducing network input and output since there is less data to be transmitted.
- by saving any processing time for non-trivially mapped columns. For example, if a column uses converters, composers, and transformations to partially inject the input record.
- by eliminating unnecessary firing of update triggers. If a CMP bean field is not changed, any trigger depending only on the corresponding column is not fired.

Although partial update improves performance, it can adversely affect performance as follows:

- If you enable partial update for a bean that your application modifies several different combinations of columns during the same time span, the prepared statement cache maximum for the connection is reached very quickly. As a result, statement handles are evicted from the cache based on least recent usage. This results in statements being prepared repeatedly, decreasing performance for all CMP functions, not just limited to the ejbStore method.
- Partial update query templates cached in the function set increase memory use. The increase is linear relative to the number of fields in the CMP bean for which the partial update access intent option is turned on.

- The PartialOperation persistent option, when used in combination with the Batch Update persistent option, affects the performance of the batch update because each partial query is different. There is an execution time cost incurred for generating a partial update query string dynamically. Since query fragments are stored for each column, the execution cost to assemble the query fragments is linear, based on the number of CMP bean fields dirtied.
- There are condition checks for each CMP field, for example, to inspect the dirty flags and to execute the preparedStatement setXXX method calls.

Considerations for using partial update

The performance gains you hope to achieve should be weighed against the possible instances where degradation can occur. You can use the following guidelines to help you make the decision.

- Partial update might not benefit an application that only involves a small table with a few columns and simple data types and no update triggers. The cost to assemble the partial query dynamically outweighs the performance gain.
- Partial update is a benefit if there is a complex data type that is not updated often. An example of a complex data type is an employee bean with a “photo” CMP attribute mapped to a BLOB OR VARGRAPHIC, or similar complex backend type, that is typically stored in a different location in the database manager implementation.
- Partial Update might benefit if there are several VARCHAR type columns and only a very few of them are updated.
- It is better not to use the partial operation if the application can randomly be updating different combinations of columns and the number of assignable columns (non-key) is greater than five. This generates different partial queries and fills up the prepared statement cache quickly. But, if the bean does not have too many columns, for example, four or less, and it has complex data types, you might consider turning partial update on, with the option of increasing the statement cache size to ensure an increased number of queries. For information on increasing the statement cache size, refer to the data source settings help.
- Partial Update is beneficial when there are update triggers needed on a subset of columns.
- Partial Update is beneficial when the table has many columns and indexes, and only a few indexes are touched by a typical update.

Restrictions

By default, batch update of update queries is disabled for all CMP beans for which partial update is enabled. In other words, partial update takes precedence over batch update. Batch update of delete and insert queries is not affected.

Batch update performance is affected when both batch update and partial update persistence options are used on the same bean, because each partial query is different. You can use the JVM property, `-Dcom.ibm.ws.pm.grouppartialupdate=true`, to group the similar partial update queries into a batch update. Grouping partial updates only helps when there are several partial queries with the same shape in a transaction. Otherwise, grouping partial updates has the opposite affect on performance. Because this setting is not on a bean level basis, you should be careful when turning it on. Because this affects all beans that have both partial update and batch update on, you must make sure that batch update of partial queries does increase performance when viewed across all the beans for which both updates are on.

To set the JVM property:

1. Open the `server.xml` file.
2. Change the value of `-Dcom.ibm.ws.pm.grouppartialupdate=true` to `-Dcom.ibm.ws.pm.grouppartialupdate=false`.

Setting partial update for container-managed persistent beans

For Enterprise JavaBeans (EJB) 2.x CMP entity beans, you can use the *partial update* feature to specify how you want to update the persistent attributes of the CMP bean to the database. This feature is provided as a bean-level persistence option, called *PartialOperation*, in the access intent policy configured for the bean.

About this task

See the topic Partial operation for container managed persistence in the assembly tool information center to learn how to complete this task with the assembly tool.

Developing EJB 3.x enterprise beans

Enterprise JavaBeans (EJB) 3.1 specification

This topic describes the Enterprise JavaBeans (EJB) 3.1 specification that is the foundation of the development and application programming model for EJB 3.1 applications. Read this topic for a brief overview of the EJB 3.1 specification.

The EJB 3.1 specification focuses on simplification and ease of use. In addition, it adds many new features to the programming model.

- *Singleton session beans* are a new type of session bean. As the name implies, only one instance of the bean exists. A Singleton is useful for storing data that is shared by different parts of an application. Data concurrency might be controlled by either the container or the application itself.
- *Non-persistent EJB Timers* are similar to the persistent EJB Timers that existed before EJB 3.1, except that they exist only in memory and are not stored in a database. Non-persistent timers are useful for scenarios where it is not desirable to retry missed events.
- *Automatically created EJB Timers* are created automatically when the application starts, and they are removed automatically when the application is uninstalled. Automatically created timers might be either persistent or non-persistent. Automatically created timers are useful because they remove the need for the application or an administrator to explicitly create and remove the timers.
- *Calendar based timer expressions* allow developers to specify a timeout schedule using a calendar-based syntax that closely resembles the UNIX Cron functionality. Calendar-based expressions are useful because they make it much easier to specify and understand the timeout schedule for a timer.
- *Asynchronous method invocation* allows applications to run multiple chunks of work in parallel. Asynchronous methods are useful from a performance perspective because work loads are not single threaded, and they are also useful from a simplification perspective because the application programmer is shielded from the complexities associated with multithreaded programming.
- The *No-Interface Local View* further simplifies the plain old Java objects (POJO) programming model. With the No-Interface Local View, EJBs are no longer required to have a bean interface.
- The *embeddable EJB container* allows developers to unit test their EJB function in a Java SE environment. The embeddable EJB container is useful because it allows developers to test EJB function quickly and easily in their personal sandbox environment, and it removes the need to install the EJBs into an application server.
- *Packaging EJB content in WAR modules* allows both web and EJB content to be physically combined into the same module. This packaging option is useful because it might simplify the assembly and installation of the application, and simplifying the interaction between the web and EJB components that are collocated in the same module.

Enterprise JavaBeans (EJB) 3.0 specification

This topic describes the Enterprise JavaBeans (EJB) 3.0 specification that is the foundation of the development and application programming model for the EJB 3.0 applications. Read this topic for a brief overview of the EJB 3.0 specification.

The EJB 3.0 specification has justifiably been called the most important upgrade to the Java Platform, Enterprise Edition 5 (Java EE 5) programming model. The EJB 3.0 specification represents simplification and streamlining of the business logic and persistence programming models used in Java EE. The ultimate source of information is the specification, which is available on the Oracle website.

While the Java Persistence API (JPA) replacement is called an entity class, it should not be confused with entity enterprise beans. A JPA entity is not an enterprise bean and is not required to run in an EJB container.

The EJB 3.0 specification is organized into three areas:

- EJB core contracts and requirements
- EJB 3.0 simplified application programming interface (API)
- JPA

The EJB core contracts and requirements define the service provider interfaces (SPIs) between the enterprise bean instance and the enterprise bean container. This part of the specification also includes the APIs between the enterprise bean provider and the enterprise bean container, protocols, component and container contracts, system level issues, infrastructure services that are provided by the container to the bean and other information about development packaging and deployment for session, message-driven, and entity beans.

The EJB 3.0 simplified API provides information about simplifying EJB APIs and SPIs that exist from previous EJB specification versions.

The JPA document introduces the Plain Old Java Object (POJO)-style persistent entity development guidelines.

Another good source for EJB 3.0 information is Mastering Enterprise JavaBeans 3.0, Fourth Edition. This edition features chapters on session beans and message-driven beans, EJB, and Java EE integration and advanced persistence concepts. Also included is coverage of the JPA and POJO using entities with the EJB programming model.

Application exceptions

Application exceptions alert the client of application specific or business logic issues; they do not report system level exceptions. This topic includes a brief overview of how application exceptions are defined and examples of the `@ApplicationException` annotation and corresponding `application-exception` deployment descriptor element.

Definition of Application Exception

Bean providers define application exceptions along with the business logic of an application. Unlike system exceptions, application exceptions are not used to report system-level errors. Instead, business methods use application exceptions to notify the client of application-level activity that might cause errors; for example, invalid input argument values to a business method. In most cases, clients can return to normal processing after experiencing application exceptions

You can define application exceptions in the throws clause of a method. The method that you use to define the application exception can be that of a business interface, no-interface view, home interface, component interface, message listener interface, or web service endpoint of the enterprise bean. When you define the exception, remember that application exceptions can be:

- A subclass, either direct or indirect, of the java.lang.Exception exception, which renders a *checked exception*
- A subclass of the java.lang.RuntimeException exception, which renders an *unchecked exception*

Attention: You cannot define an application exception as a subclass of java.rmi.RemoteException because this exception and its subclasses are for system-level problems.

The following standard application exceptions and their subclasses are used to report errors to the client:

- javax.ejb.CreateException
- javax.ejb.RemoveException
- javax.ejb.FinderException

The previous application exceptions are defined in the create, remove, and finder methods of the EJBHome interface, the EJBLocalHome interface, or both. These interfaces come from components that are written to the EJB 2.1 client view.

In the Enterprise Java beans 3.0 specification, the @ApplicationException annotation had only one optional parameter of rollback. The corresponding application-exception element had only one optional subelement, rollback, that you set to true or false. The rollback parameter/subelement is used to specify whether the transaction is marked for rollback. By default this value is false. Inheritance of an application exception could not be specified and was not given an explicit default value in the EJB 3.0 specification. The product implementation of the EJB 3.0 specification did not provide application exception inheritance, unless an application exception was defined on the *throws* clause of a business method of a bean. In contrast, the EJB 3.1 specification has introduced the optional `inherited` parameter to the @ApplicationException annotation and the optional `inherited` subelement to the corresponding application-exception deployment descriptor element. By default, marking an exception as an application exception causes all subclasses of that exception to also be application exceptions (that is, `inherited=true`). You can disable the inheriting behavior by setting the `inherited` parameter of the @ApplicationException to false. Likewise, you can disable the inheriting behavior by setting the `inherited` subelement of the application-exception element in the deployment descriptor to false. For more information about application exceptions, refer to section 14.1.1 of the EJB 3.1 specification.

Inherited application exceptions using annotations:

```
import javax.ejb.ApplicationException;

@ApplicationException(inherited=true, rollback=true)
public class RTExceptionA extends RuntimeException{

   //RTExceptionA

}
public class RTExceptionB extends RTExceptionA{

   //RTExceptionB

}
import javax.ejb.ApplicationException;

@ApplicationException(inherited=false, rollback=false)
public class RTExceptionC extends RTExceptionB{

   //RTExceptionC

}
public class RTExceptionD extends RTExceptionC{

   //RTExceptionD

}
```

The previous example yields the following results:

RTExceptionA is an application exception with transaction rollback.

RTExceptionB is an application exception with transaction rollback.

RTExceptionC is an application exception without transaction rollback.

RTExceptionD is not an application exception.

Inherited application exceptions using XML:

```
public class RTExceptionA extends RuntimeException{

   //RTExceptionA

}
public class RTExceptionB extends RTExceptionA{

   //RTExceptionB

}
public class RTExceptionC extends RTExceptionB{

   //RTExceptionC

}
public class RTExceptionD extends RTExceptionC{

   //RTExceptionD

}
```

```
<!-- Example ejb-jar.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
    metadata-complete="true" version="3.1">
    <assembly-descriptor>
        <application-exception>
            <exception-class>myXML.example.package.RTExceptionA</exception-class>
            <rollback>true</rollback>
            <inherited>true</inherited>
        </application-exception>
        <application-exception>
            <exception-class>myXML.example.package.RTExceptionC</exception-class>
            <rollback>false</rollback>
            <inherited>false</inherited>
        </application-exception>
    </assembly-descriptor>
</ejb-jar>
```

As in the annotation version, the previous example yields the following results:

RTExceptionA is an application exception with transaction rollback.

RTExceptionB is an application exception with transaction rollback.

RTExceptionC is an application exception without transaction rollback.

RTExceptionD is not an application exception.

Remember: You can use the `rollback` and `inherited` subelements of the `application-exception` to explicitly override the `rollback` and `inherited` attribute values that were specified or implicitly set by the `@ApplicationException` annotation.

When you specify an exception on the throws clause of a business method of a bean, the resulting checked exception is an application exception. All subclasses of this application exception are also application exceptions. No option is available to disable this inheriting behavior of checked application exceptions. You can use the `inherited` element to determine the rollback value of the checked application

exception subclasses. If the `inherited` element of the checked application exception is set to `true` and its `rollback` element is set to `true`, then the subclasses of that checked application exception inherit the `rollback = true` value.

Getting EJB 3.0 application exception inheritance behavior:

You have the following options if you have an existing EJB 3.0 application and you want to continue having the application exception inheritance behavior be false, which means that the subclasses of an application exception are not application exceptions themselves.

- You can modify the exception's `@ApplicationException` annotation of the exception by adding the `inherited` attribute and setting it to `false`.
- You can add a version 3.1 deployment descriptor and explicitly set the `inherited` subelement of the `application-exception` element to `false`. If you have an existing version 3.0 deployment descriptor you must migrate to a version 3.1 deployment descriptor and XSD schema and set the `inherited subelement of the application-exception` element to `false`.

How to get EJB 3.0 application exception inheritance behavior with annotations:

Suppose that you previously had the following code:

```
import javax.ejb.ApplicationException;

@ApplicationException()
public class EJB30_RTException extends RuntimeException{

   //EJB30_RTException, in EJB 3.0 subclasses were not application exceptions

}
```

You must modify the `@ApplicationException` annotation to include the `inherited=false` attribute:

```
import javax.ejb.ApplicationException;

@ApplicationException(inherited=false)
public class EJB30_RTException extends RuntimeException{

   //EJB30_RTException,  now you must explicitly set inherited to false to disable inheritance

}
```

How to get EJB 3.0 application exception inheritance behavior using XML:

Suppose that you previously had the following code:

```
<!-- Example ejb-jar.xml -->

<?xml version="1.0" encoding="UTF-8"?>
  <ejb-jar id="ejb-jar_ID" xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
      metadata-complete="true" version="3.0">
    <assembly-descriptor>
        <application-exception>
            <exception-class>myXML.example.package.EJB30_RTException</exception-class>
        </application-exception>
    </assembly-descriptor>
</ejb-jar>
```

You must modify your code to include the `inherited` element set to `false` as well as migrate to a version 3.1 deployment descriptor and XSD schema:

```
<!-- Example ejb-jar.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar id="ejb-jar_ID" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
```

```
    metadata-complete="true" version="3.1">
    <assembly-descriptor>
        <application-exception>
            <exception-class>myXML.example.package.EJB30_RTException</exception-class>
            <rollback>false</rollback>
            <inherited>false</inherited>
        </application-exception>
    </assembly-descriptor>
</ejb-jar>
```

EJB 3.x module considerations

When using Enterprise JavaBeans (EJB) 3.x modules, keep in mind the following considerations.

Version 8.0 does not support 1.x and 2.x entity beans in EJB 3.x-level modules

IBMWebSphere Application Server Version 8.0 does not support the use of 1.x and 2.x bean managed persistence (BMP) and container managed persistence (CMP) entity beans in EJB 3.x-level modules. EJB entity beans can be used on V8.0, but they must be packaged in an EJB 2.1 or earlier-level module.

Java Platform, Enterprise Edition (Java EE) applications that are packaged with EJB entity beans in EJB 3.x-level modules fail to install on Version 8.0.

An EJB Java archive (JAR) file is considered to be an EJB 3.x module when either of the following are true:

- The EJB JAR file contains configuration data in an `ejb-jar.xml` file with an EJB 3.0 or EJB 3.1 header specification. For example:

```
<ejb-jar id="ejb-jar_ID" version="3.0"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
```

- The EJB JAR file contains beans with EJB 3.x-style source annotations that provide configuration data and does not contain an `ejb-jar.xml` deployment descriptor file.

You need to repackage your EJB 3.x modules using EJB 2.x and earlier modules. Otherwise, the installation of any applications that contain entity beans fail.

Annotations

Consider using annotations versus deployment descriptors, or both. See the topic EJB 3.x metadata annotations for more information about annotations.

EJB module

WebSphere Application Server Version 8.0 supports EJB module Java archive (JAR) files with an `ejb-jar.xml` deployment descriptor declared at the 1.1, 2.0, 2.1, 3.0, or 3.1 level, or with no `ejb-jar.xml` deployment descriptor present. If no deployment descriptor is present, the EJB module is assumed to be at the 3.0 level or greater.

EJB modules that contain EJB 3.x beans must be declared to be at the EJB 3.0 or EJB 3.1 level. This can be accomplished either by setting the `ejb-jar.xml` deployment descriptor level to 3.0 or 3.1, or ensuring that the module does not contain an `ejb-jar.xml` deployment descriptor. If the module level is 2.1 or earlier, no EJB 3.x-specific functions such as annotation scanning or resource injection is performed.

Entity beans are not supported in EJB 3.x level modules. You must place any entity beans in EJB modules at the 2.1 or earlier level.

If you want to package an EJB 3.x module with a deployment descriptor, there are several ways to do it. Package an EJB 3.x module with an EJB 3.x style session and message-driven beans exclusively; with an EJB 2.1 style session and message-driven beans exclusively, or a combination of 2.1 and 3.x style beans. The XML deployment descriptor must be a Version 3.0 or 3.1 deployment descriptor. It is required that 2.1 entity beans are packaged in modules with 2.1 deployment descriptors. EJB modules that contain EJB 3.x beans must be at the EJB 3.0 or EJB 3.1 specification level when running on the product. Prepare the EJB module to support EJB 3.x beans, by setting the `ejb-jar.xml` deployment descriptor level to 3.0 or 3.1, or you can make sure that the module does not contain an `ejb-jar.xml` deployment descriptor. If the module level is EJB 2.1 or earlier, no EJB 3.x functions, including annotation scanning or resource injection is performed at run time.

Java EE application client module

The product provides support for Java EE application client modules. Additionally, it supports injection of EJB references into client components if the injection is defined through the @EJB annotation.

Attention: EJB 3.x does not support the injection of an enterprise bean that creates an enterprise bean of itself. Do not inject an enterprise bean that creates an enterprise bean of itself.

Defining an ejb-ref reference to an EJB 3.x business interface from a Java EE client component descriptor

It is possible to define an ejb-ref from an `application-client.xml` descriptor that points to an EJB 3.x business interface. EJB 3.x business interfaces are accessed directly without the use of a home, yet the ejb-ref element in Java EE requires that a home interface type is specified. Therefore, you must include the <home></home> stanza in the ejb-ref definition, but specify a null value as shown in the example. For the value of the <remote> stanza, specify the EJB 3.x business interface class name. Finally, when you set the binding value, either during application installation or through tooling, specify the location where the EJB 3.x business interface was bound.

For example, the ejb-ref in your client component `application-client.xml` file looks like the following code sample:

```
<ejb-ref id="EJBRef_1">
 <ejb-ref-name>java_comp-env_name_of_ref</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home></home>
 <remote>com.ejbs.business.interface.class.name</remote>
</ejb-ref>
```

The corresponding section of the `ibm-application-client-bnd.xmi` file looks like the following code sample. A default EJB binding pattern is used here; the default EJB binding conventions are described in the topic, EJB 3.x applications binding support.

```
<ejbRefBindings xmi:id="EjbRefBinding_1"
         jndiName=EJB3App/EJB3Mod.jar/MyBean##com.ejbs.business.interface.class.name">
 <bindingEjbRef href="application-client.xml#EjbRef_1"/>
</ejbRefBindings>
```

EJB metadata annotations

Annotations enable you to write metadata for Enterprise JavaBeans (EJB) inside your source code. You can use them instead of extensible markup language (XML) deployment descriptor files. Annotations can also be used *with* descriptor files.

If you installed the Feature Pack for EJB 3.0, the default was to scan annotations during the installation of an EJB 3.0 module. For WebSphere Application Server, Version 7.0 and later, the default is not to scan pre-Java EE 5 modules during the application installation or at server startup

To preserve backward compatibility with both the Feature Pack for EJB 3.0 and the Feature Pack for Web Services, you have a choice whether to scan legacy web modules for additional metadata. A server level

switch is defined for each feature pack scan behavior. If the default is not appropriate, the switch must be set on each server and administrative server that requires a change in the default. The switches are server custom properties com.ibm.websphere.webservices.UseWSFEP61ScanPolicy={true|false} and com.ibm.websphere.ejb.UseEJB61FEPScanPolicy={true|false}. To define these properties in the administrative console click **Application servers** > **server name** > **Process definition** > **Java Virtual Machine** > **Custom properties**.

The product also provides default values for most of the EJB annotations it uses. In many cases, omitting an annotation implies that you want to use the default value.

Usually, annotations are found in the *javax.ejb* and *javax.persistence* packages.

Table 47. Annotation types. Annotation types

Annotation type
AccessTimeout
ExcludeDefaultInterceptors
AfterBegin
AfterCompletion
ApplicationException
AroundInvoke
Asynchronous
BeforeCompletion
ConcurrencyManagement
DependsOn
EJB
EJBs
ExcludeDefaultInterceptors
ExcludeDefaultInterceptors
Init
Interceptors
Local
Lock
LocalBean
LocalHome
MessageDriven
PersistenceUnit
PostActivate
PostConstruct
PreDestroy
PrePassivate
Remote
RemoteHome
Remove
Resource
Schedule

Table 47. Annotation types (continued). Annotation types

Annotation type
Schedules
Singleton
Startup
StatefulTimeout
Stateless
Timeout
TransactionAttribute
TransactionManagement
TransactionManagement

EJB 3.x interceptors

An interceptor is a method that is automatically called when the business methods of an Enterprise JavaBeans (EJB) are invoked or lifecycle events of an EJB occur.

There are three kinds of interceptor methods: business method interceptors, timeout method interceptors (which are new in EJB3.1), and lifecycle callback interceptors. Business method interceptors are invoked around the call to a business method. Timeout method interceptors are invoked around the call to an EJB timeout method. Lifecycle callback interceptors are called around one of the PostConstruct, PreDestroy, PrePassivate, or PostActivate lifecycle events. For each interceptor type, an individual class can only declare one interceptor method. However, each class in a class hierarchy can declare an interceptor method for each interceptor type. If an interceptor method in a subclass overrides the same method in a super-class, only the method in the subclass might be invoked.

Interceptor methods are permitted to access and call all resources and components that the associated method is allowed to call. Additionally, interceptor methods execute with the same transaction and security context as the associated method. Except for singleton session beans, lifecycle interceptor methods are executed with local transaction containment (LTC).

You can declare interceptor methods directly in the EJB class or in a separate interceptor class. To declare interceptor methods in a separate class, you must bind the interceptor class to the EJB using either an annotation or XML. Use the following example to declare interceptors using the annotation:

```
@Interceptors({ClassInterceptor1.class, ClassInterceptor2.class})
public class TestBean { /* ... */ }

@Interceptors({ClassInterceptor1.class})
public class TestBean2 {
 @Interceptors({MethodInterceptor1.class, MethodInterceptor2.class})
 public void businessMethod() { /* ... */ }
}
```

Use the following example to declare interceptors using the deployment descriptor:

```
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>TestBean</ejb-name>
      <interceptor-class>ClassInterceptor1</interceptor-class>
      <interceptor-class>ClassInterceptor2</interceptor-class>
    </interceptor-binding>

    <interceptor-binding>
      <ejb-name>TestBean2</ejb-name>
      <interceptor-class>ClassInterceptor1</interceptor-class>
    </interceptor-binding>
```

```
    <interceptor-binding>
      <ejb-name>TestBean2</ejb-name>
      <interceptor-class>MethodInterceptor1</interceptor-class>
      <interceptor-class>MethodInterceptor2</interceptor-class>
      <method>
        <method-name>businessMethod</method-name>
      </method>
    </interceptor-binding>
  </assembly-descriptor>
```

You can exclude class-level interceptors from a method by using either the ExcludeClassInterceptors annotation or the exclude-class-interceptors element in the deployment descriptor. Use the following example to exclude the interceptor, ClassInterceptor, from the method, businessMethod.

```
@Interceptors({ClassInterceptor1.class})
public class TestBean2 {
 @ExcludeClassInterceptors
 public void businessMethod() { /* ... */ }

 public void businessMethodWithClassInterceptor1() { /* ... */
}
```

Use the following example to exclude the interceptor from the method using the deployment descriptor:

```
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>TestBean2</ejb-name>
      <exclude-class-interceptors>true</exclude-class-interceptors>
      <method>
        <method-name>businessMethod</method-name>
      </method>
    </interceptor-binding>
  </assembly-descriptor>
```

Interceptor methods can have public, protected, package private, or private visibility. Interceptor methods must not be final or static. Business method interceptors and timeout method interceptors must have a return type of java.lang.Object, a single parameter of javax.interceptor.InvocationContext, and a single throws clause type of java.lang.Exception. All lifecycle interceptors must have a return type of void and must not have a throws clause. Lifecycle interceptors declared directly on the EJB class must not have parameters; lifecycle interceptors declared on an EJB superclass or on an interceptor class must have a single parameter of javax.interceptor.InvocationContext. Timeout interceptor methods and lifecycle interceptor methods must not throw application exceptions.

You can use the InvocationContext parameter of an interceptor method to get information about the method being invoked. The getTarget method returns the bean instance being invoked. The getTimer method is applicable to timeout method interceptors only, and it returns the timer being executed. The getMethod method returns the business interface method that is being invoked. The getParameters method returns the parameters being passed to the business method, and the setParameters method allows the parameters to be modified. The getContextData method returns the data association with the method being invoked. Finally, the proceed method invokes either the next interceptor or the target method.

You can declare interceptor methods using either annotations or XML. To declare an interceptor method using an annotation, place the appropriate AroundInvoke, AroundTimeout, PostConstruct, PreDestroy, PrePassivate, or PostActivate annotation on the interceptor method. Use the following example to declare a business method interceptor, a timeout method interceptor, and a PostConstruct lifecycle interceptor method on an EJB class using annotations.

```
@Interceptors({ClassInterceptor.class})
public class TestBean {
 @PostConstruct
 private void beanPostConstruct() { /* ... */ }
```

```
 @AroundInvoke
 protected Object beanAroundInvoke(InvocationContext ic) throws Exception {
  return ic.proceed();
 }

        @AroundTimeout
        protected Object beanAroundTimeout(InvocationContext ic) throws Exception {
  return ic.proceed();
        }
}
```

Use the following example to declare the same interceptor methods on an interceptor class.

```
public class ClassInterceptor {
 @PostConstruct
 private void interceptorPostConstruct(InvocationContext ic) {
  try {
   ic.proceed();
  } catch (Exception ex) { /* ... */ }
 }

 @AroundInvoke
 protected Object interceptorAroundInvoke(InvocationContext ic) throws Exception {
  return ic.proceed();
 }

 @AroundTimeout
 protected Object interceptorAroundTimeout(InvocationContext ic) throws Exception {
  return ic.proceed();
 }
}
```

Alternatively, you can declare an interceptor method in the deployment descriptor with the around-invoke, around-timeout, post-construct, pre-destroy, pre-passivate, and post-activate elements. Use the following example to declare a business method interceptor, a timeout method interceptor, and a PostConstruct lifecycle interceptor method on an EJB class and an interceptor class using the deployment descriptor.

```
  <enterprise-beans>
    <session>
      <ejb-name>TestBean</ejb-name>
      <around-invoke>
        <method-name>beanAroundInvoke</method-name>
      </around-invoke>
      <around-timeout>
        <method-name>beanAroundTimeout</method-name>
      </around-timeout>
      <post-construct>
        <lifecycle-callback-method>beanPostConstruct</lifecycle-callback-method>
      </post-construct>
    </session>
  </enterprise-beans>

  <interceptors>
    <interceptor>
      <interceptor-class>ClassInterceptor</interceptor-class>
      <around-invoke>
        <method-name>interceptorAroundInvoke</method-name>
      </around-invoke>
      <around-timeout>
        <method-name>interceptorAroundTimeout</method-name>
      </around-timeout>
      <post-construct>
        <lifecycle-callback-method>interceptorPostConstruct</lifecycle-callback-method>
      </post-construct>
    </interceptor>
  </interceptors>
```

```
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>TestBean</ejb-name>
      <interceptor-class>ClassInterceptor</interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
```

You can also declare interceptor methods on super-classes. Use the following example to declare a PostActivate interceptor on a bean super-class using annotations:

```
public class TestBean extends BeanSuperClass { /* ... */ }

public class BeanSuperClass {
 @PostActivate
 private void beanSuperClassPostActivate(InterceptorContext ic) {
  try {
   ic.proceed();
  } catch (Exception ex) { /* ... */ }
 }
}
```

Use the following example to declare the same interceptor method on a superclass of an interceptor class using annotations:

```
public class ClassInterceptor extends InterceptorSuperClass { /* ... */ }

public class InterceptorSuperClass {
 @PostActivate
 private void interceptorSuperClassPostActivate(InterceptorContext ic) {
  try {
   ic.proceed();
  } catch (Exception ex) { /* ... */ }
 }
}
```

You can also declare the same interceptor methods using the deployment descriptor. Use the following example to declare an interceptor method on the superclasses of a bean and interceptor class:

```
<enterprise-beans>
  <session>
    <ejb-name>TestBean</ejb-name>
    <post-activate>
      <class>BeanSuperClass</class>
  <lifecycle-callback-method>beanSuperClassPostActivate</lifecycle-callback-method>
    </post-activate>
  </session>
 </enterprise-beans>

 <interceptors>
   <interceptor>
    <interceptor-class>ClassInterceptor</interceptor-class>
    <post-activate>
      <class>InterceptorSuperClass</class>
  <lifecycle-callback-method>interceptorSuperClassPostActivate</lifecycle-callback-method>
    </post-activate>
   </interceptor>
 </interceptors>

 <assembly-descriptor>
   <interceptor-binding>
     <ejb-name>TestBean</ejb-name>
     <interceptor-class>ClassInterceptor</interceptor-class>
   </interceptor-binding>
 </assembly-descriptor>
```

You can declare default interceptors that apply to all session and message-driven beans in a module. Default interceptors can only be declared in the deployment descriptor, and they are specified using an ejb-name of "*". Use the following example to declare a default interceptor.

```
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>DefaultInterceptor</interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
```

You can exclude default interceptors from a specific class or method by using either the ExcludeDefaultInterceptors annotation or the exclude-default-interceptors element in XML. Use the following examples to exclude default interceptors using the annotation:

```
@ExcludeDefaultInterceptors
public class TestBean { /* ... */ }

public class TestBean2 {
 @ExcludeDefaultInterceptors
 public void businessMethod() { /* ... */ }
}
```

Use the following example to excludes default interceptors using the deployment descriptor:

```
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>TestBean</ejb-name>
      <exclude-default-interceptors>true</exclude-default-interceptors>
    </interceptor-binding>

    <interceptor-binding>
      <ejb-name>TestBean2</ejb-name>
      <exclude-default-interceptors>true</exclude-default-interceptors>
      <method>
        <method-name>businessMethod</method-name>
      </method>
    </interceptor-binding>
  </assembly-descriptor>
```

When interceptors are invoked for a method, default interceptor classes are invoked first, class-level interceptors are invoked next, and interceptors methods from the EJB class are invoked last. For a single interceptor class hierarchy, interceptor methods are always invoked on the most general super-class first. The default and class-level interceptor classes are invoked in the order specified in the deployment descriptor or the Interceptors annotation. You can override this ordering by specifying the complete list of default and class-level interceptors in the interceptor-order element in the deployment descriptor.

```
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>TestBean</ejb-name>
      <!--
        The default ordering would be:
   1. DefaultInterceptor
   2. ClassInterceptor1
   3. ClassInterceptor2

 The following stanza overrides the default ordering.
      -->
      <interceptor-order>
        <interceptor-class>ClassInterceptor2</interceptor-class>
 <interceptor-class>DefaultInterceptor</interceptor-class>
 <interceptor-class>ClassInterceptor1</interceptor-class>
      </interceptor-order>
    </interceptor-binding>
  </assembly-descriptor>
```

Create stubs command

The createEJBStubs command creates stub classes for remote interfaces of Enterprise JavaBeans (EJB) Version 3.0 beans packaged in Java archive (JAR) or Enterprise archive (EAR) files. It also provides an option to create a single stub class from an interface class located in a directory or a JAR file. Several command options are provided to package the generated stub classes in different ways. See the Syntax and Examples sections later in this topic for more details.

This command is found in the <WAS_HOME>/bin directory as:
- createEJBStubs.bat - Windows platforms
- createEJBStubs.sh - Unix based platforms
- createEjbStubs - iSeries platform

The command searches the input JAR or EAR file, looking for EJB version 3.0 modules that contain beans with remote interfaces. When remote interfaces are found, the corresponding stub classes are generated and packaged according to the command options specified. In the case where the input specified is a single interface class, the tool assumes this class is an EJB version 3.0 remote interface class and generates a remote stub class.

For many client-side scenarios, the WebSphere Application Server Just-In-Time (JIT) deployment feature dynamically generates the RMI-IIOP stub classes that are required for invocation of remote EJB 3.0 business interfaces. However, there are some scenarios where the JIT deploy environment is not available to dynamically generate these classes. In these scenarios, the createEJBStubs command must be used instead to generate and embed the client-side stub class files in your client application. If your client environment is one of the following, use the createEJBStubs command:
- “Bare”Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.
- A WebSphere Application Server container (web container, EJB container, or application client container) from a version earlier than version 7, or without the Feature Pack for EJB 3.0 applied.
- Non-WebSphere Application Server environments.

Syntax

createEJBStubs input_ class_ name | input_JAR_name | input_EAR_name [**-help**] [**-newfile [new_file]**] [**-updatefile [update_file]**] [**-quiet**] [**-verbose**] [**-logfile log_file**] [**-appendlog**] [**-cp class_path**] [**-trace**]

createEJBStubs

This is the command to create EJB stub classes for a single interface class file, a JAR file, or an EAR file. When invoked without any arguments, or only –help, the createEJBStubs command displays a list of options that can be specified, and a list of example invocations with detailed explanations.

input_class_name or input_EAR_name or input_JAR_name

The first parameter is a required element for the command. It must contain the source class, JAR, or EAR file to process.

This parameter may be the fully qualified name of a single interface class (e.g. com.ibm.myRemoteInterface). Note that the package name segments are separated by “.” characters, no path name proceeds the class name, and the “.class”extension is not included. For this interface class input, you must use the class path option (e.g. –cp my_path, or –cp my_path/my_interfaces.jar) to specify where the interface class will be found. The generated stub class will be placed in the package-defined directory structure, starting with the current directory where the command is invoked.

This parameter may also be a JAR or EAR file. In this case the path must be specified (e.g. my_path/my_Server_App.ear). The generated stub classes will be placed in the same module or

modules with the beans, or in the same module or modules with the remote interface classes, depending on whether the –updatefile option is specified. Details follow later in this section.

-help Provides the command syntax, including a list of options that can be specified, and example invocations with detailed explanations.

-newfile [new_file]
Requests that a new file is generated containing the original files in the input JAR or EAR plus the stub classes. When this option is not specified, the stubs are written back into the original JAR or EAR file. If this option is specified, but the new_file name is not provided, a new file name is constructed by appending the input JAR or EAR file name with "_withStubs". This option is not allowed when the first input parameter is an interface class.

-updatefile [update_file]
Requests that a second file (e.g. in addition to the input file) is updated with stub classes. This option also provides a different packaging behavior. The stub classes are packaged in the same module or modules as the remote interface classes. By contrast, when this option is not specified, the stub classes are packaged in the same module or modules with the bean classes. If this option is specified, but the update_file name is not provided only the original JAR or EAR file is updated with stub classes. This option is not allowed when the first input parameter is an interface class.

-quiet Requests the suppression of messages. The -quiet option cannot be specified with either the –verbose or the –trace options. Error messages are still displayed.

-verbose
Requests that additional informational messages be output. The -verbose option cannot be specified with either the -quiet or the –trace options.

-logfile log_file
Requests that messages be printed to a log file in addition to the console. If this option is specified, the log_file name must also be provided.

-appendlog
Requests that messages be appended to an existing log file. If this option is specified, the -logfile option must also be specified.

-cp class_path
Requests that the classloader includes the specified the class path where additional class or jar files are located, which are necessary for the remote interface classes to be loaded. The class path may include multiple segments where each path is separated from a previous path by the default path separator character of the operating system. Each path can specify either a JAR file, or a directory. If this option is specified, the class_path name must also be provided.

-trace Request that detailed trace output be generated. This is intended to collect information for use by IBM service to resolve problems. The trace output is English-only. This option cannot be specified with either the –quiet or the –verbose options.

Examples

```
createEJBStubs com.ibm.myRemoteInterface -cp my_path
```

Generate the stub class for one remote interface class and place it in the package-defined directory structure, starting at the current directory. The my_path directory will be used as the class path. If the remote interface class to process is in a JAR file, the –cp my_path/my_interfaces.jar syntax must be used for the class path specification.

```
createEJBStubs my_path/my_beans.jar -newfile –quiet
```

Generate the stub classes for all level 3.0 enterprise beans in the my_beans.jar file that have remote interfaces. Both the generated stub classes and the original JAR file contents are packaged into a new JAR file named “my_beans_withStubs.jar” because the optional new_file name parameter is not specified along with the –newfile option. Output messages are suppressed except for error notifications.

```
createEJBStubs my_path/my_Server_App.ear -logfile myLog.out
```

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are placed into the original EAR file because the –newfile option is not specified. The stub classes are packaged into the same module or modules as the bean classes because the –updatefile option is not specified. Messages are written to both the myLog.out log file and the command window.

```
createEJBStubs my_path/my_Server_App.ear -updatefile my_path/my_Client_interfaces.jar
```

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are placed into both the original EAR file and the my_Client_interfaces.jar file. The stub classes are packaged into the same module or modules as the remote interface classes because the –updatefile option is specified.

```
createEJBStubs my_path/my_Server_App.ear –updatefile
```

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are only placed into the original EAR file because the optional update_file name parameter is not provided with the –updatefile option. The stub classes are packaged into the same module or modules as the remote interface classes because the -updatefile option is specified.

Create stubs command

The createEJBStubs command creates stub classes for remote interfaces of Enterprise JavaBeans (EJB) Version 3.x beans packaged in Java archive (JAR), web application archive (WAR), or enterprise archive (EAR) files. It also provides an option to create a single stub class from an interface class located in a directory or a JAR file. Several command options are provided to package the generated stub classes in different ways. See the following syntax and example sections for more details.

This command is found in the <*WAS_HOME*>/`bin` directory as:
- createEJBStubs.bat - Windows platforms
- createEJBStubs.sh - UNIX based platforms
- createEjbStubs - iSeries platform

The command searches the input JAR, WAR, or EAR file, looking for EJB version 3.x modules that contain beans with remote interfaces. When remote interfaces are found, the corresponding stub classes are generated and packaged according to the command options specified. In the case where the input specified is a single interface class, the tool assumes that this class is an EJB version 3.x remote interface class and generates a remote stub class.

For many client-side scenarios, the WebSphere Application Server Just-In-Time (JIT) deployment feature dynamically generates the RMI-IIOP stub classes that are required for invocation of remote EJB 3.x business interfaces. However, there are some scenarios where the JIT deploy environment is not available to dynamically generate these classes. In these scenarios, the createEJBStubs command must be used instead to generate and embed the client-side stub class files in your client application. If your client environment is one of the following, use the createEJBStubs command:
- "Bare" Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.

- A WebSphere Application Server container (web container, EJB container, or application client container) from a version earlier than version 7, or without the Feature Pack for EJB 3.0 applied.
- Non-WebSphere Application Server environments.

The JVM running the createEJBStubs command must have the java.io.tmpdir system property defined. The property must point to a readable and writable directory that exists.

In addition to creating stubs for bean content packaged in EAR and JAR files, the command also creates stubs for bean content that is packaged in a WAR file. If an EAR file is specified as the input file, and that EAR file contains a WAR file with remote interfaces, then stubs are generated for those interfaces.

Stubs are placed into the WAR file in the same location as the remote interface or bean class that they correspond to. If the remote interface or bean class that they correspond to is placed loosely in the WEB-INF/classes directory structure, then the stub is placed there as well. If the remote interface or bean class is packaged inside of a JAR file in the WEB-INF/lib directory, then the stub is inserted into that same JAR file. If a stub is generated for a remote interface that is not packaged inside the WAR file, then it is placed loosely in the WEB-INF/classes directory structure.

The createEJBStubs command is not supported for use with a 2.x or 1.x EJB module packaged inside a stand-alone JAR file. The stubs for a 2.x or 1.x EJB module packaged inside a stand-alone JAR file must be generated using the EJBDeploy tool instead. If the createEJBStubs command is run against a 2.x or 1.x EJB module packaged inside a stand-alone JAR file, then the command issues a CNTR9241I message, and does not generate any stubs.

However, the createEJBStubs command is supported for use with 2.x or 1.x EJB modules packaged inside a WAR file. In this case, the createEJBStubs command must be used instead of the EJBDeploy tool.

A client component uses these stubs to communicate with the EJB components running inside the server. The client component must use stub instances that were created by the correct tool. If the client component is communicating with a 2.x or 1.x EJB module packaged inside a stand-alone JAR, then the stubs must come from the EJBDeploy tool. However, if the client component is communicating with a 2.x or 1.x EJB module packaged inside a WAR, then the stubs must come from the createEJBStubs command. A single stub instance cannot be used to communicate with both a 2.x/1.x EJB module packaged inside a stand-alone JAR, and that same 2.x/1.x EJB module packaged inside a WAR.

Syntax

createEJBStubs*input_ class_ name* | *input_JAR_name* | *input_WAR_name* | *input_EAR_name* [**-help**] [**-newfile[new_file]**] [**-updatefile[update_file]**] [**-quiet**] [**-verbose**] [**-logfile log_file**] [**-appendlog**] [**-cp class_path**] [**-trace**]

createEJBStubs

This is the command to create EJB stub classes for a single interface class file, a JAR file, a WAR file, or an EAR file. When invoked without any arguments, or only –help, the createEJBStubs command displays a list of options that can be specified, and a list of example invocations with detailed explanations.

input_class_name **or** *input_EAR_name* **or** *input_JAR_name* **or** *input_WAR_name*

The first parameter is a required element for the command. It must contain the source class, JAR, WAR, or EAR file to process.

This parameter may be the fully qualified name of a single interface class (e.g. com.ibm.myRemoteInterface). Note that the package name segments are separated by "." characters, no path name proceeds the class name, and the ".class" extension is not included. For this interface class input, you must use the class path option (e.g. –cp my_path, or –cp

my_path/my_interfaces.jar) to specify where the interface class is found. The generated stub class is placed in the package-defined directory structure, starting with the current directory where the command is invoked.

This parameter may also be a JAR, WAR, or EAR file. In this case the path must be specified (e.g. my_path/my_Server_App.ear). The generated stub classes are placed in the same module or modules with the beans, or in the same module or modules with the remote interface classes, depending on whether the –updatefile option is specified. See the following sections for more details.

-help Provides the command syntax, including a list of options that can be specified, and example invocations with detailed explanations.

-newfile [new_file]
Requests that a new file is generated containing the original files in the input JAR, WAR, or EAR plus the stub classes. When this option is not specified, the stubs are written back into the original JAR, WAR, or EAR file. If this option is specified, but the new_file name is not provided, a new file name is constructed by appending the input JAR, WAR, or EAR file name with "_withStubs". This option is not allowed when the first input parameter is an interface class.

-updatefile [update_file]
Requests that a second file (e.g. in addition to the input file) is updated with stub classes. This option also provides a different packaging behavior. The stub classes are packaged in the same module or modules as the remote interface classes. By contrast, when this option is not specified, the stub classes are packaged in the same module or modules with the bean classes. If this option is specified, but the update_file name is not provided only the original JAR, WAR, or EAR file is updated with stub classes. This option is not allowed when the first input parameter is an interface class.

-quiet Requests the suppression of messages. The -quiet option cannot be specified with either the –verbose or the –trace options. Error messages are still displayed.

-verbose
Requests that additional informational messages be output. The -verbose option cannot be specified with either the -quiet or the –trace options.

-logfile log_file
Requests that messages be printed to a log file in addition to the console. If this option is specified, the log_file name must also be provided.

-appendlog
Requests that messages be appended to an existing log file. If this option is specified, the -logfile option must also be specified.

-cp class_path
Requests that the class loader includes the specified the class path where additional class or jar files are located, which are necessary for the remote interface classes to be loaded. The class path might include multiple segments where each path is separated from a previous path by the default path separator character of the operating system. Each path can specify either a JAR file, or a directory. If this option is specified, the class_path name must also be provided.

-trace Request that detailed trace output be generated. This is intended to collect information for use by IBM service to resolve problems. The trace output is English-only. This option cannot be specified with either the –quiet or the –verbose options.

Examples

createEJBStubs com.ibm.myRemoteInterface -cp my_path

Generate the stub class for one remote interface class and place it in the package-defined directory structure, starting at the current directory. The my_path directory is used as the class path. If the remote interface class to process is in a JAR file, the –cp my_path/my_interfaces.jar syntax must be used for the class path specification.

createEJBStubs my_path/my_beans.jar -newfile –quiet

Generate the stub classes for all level 3.0 enterprise beans in the my_beans.jar file that have remote interfaces. Both the generated stub classes and the original JAR file contents are packaged into a new JAR file named “my_beans_withStubs.jar” because the optional new_file name parameter is not specified along with the –newfile option. Output messages are suppressed except for error notifications.

createEJBStubs my_path/my_Server_App.ear -logfile myLog.out

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are placed into the original EAR file because the –newfile option is not specified. The stub classes are packaged into the same module or modules as the bean classes because the –updatefile option is not specified. Messages are written to both the myLog.out log file and the command window.

createEJBStubs my_path/my_Server_App.ear -updatefile my_path/my_Client_interfaces.jar

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are placed into both the original EAR file and the my_Client_interfaces.jar file. The stub classes are packaged into the same module or modules as the remote interface classes because the –updatefile option is specified.

createEJBStubs my_path/my_Server_App.ear –updatefile

Generate the stub classes for all level 3.0 enterprise beans in the my_Server_App.ear file that have remote interfaces. The generated stub classes are only placed into the original EAR file because the optional update_file name parameter is not provided with the –updatefile option. The stub classes are packaged into the same module or modules as the remote interface classes because the -updatefile option is specified.

createEJBStubs my_path/my_beans.war

Generate the stub classes for all 3.x beans that are packaged in the WAR file and have remote interfaces. The Bean1 class is packaged loosely inside the WEB-INF/classes directory structure, and therefore, the corresponding stub is placed there as well. Likewise, the Bean2 class is packaged inside of the myEJB.jar file in the WEB-INF/lib directory, and therefore, the corresponding stub is inserted into that JAR file.

Developing entity beans

Defining data sources for entity beans

An application that is installed on an application server must have bindings defined before you can start the application. The Enterprise JavaBeans (EJB) references and resource references that are defined in the application must be bound to the actual enterprise beans or resources that are defined in the application server.

Before you begin

Create a data source or JDBC resource and give it a Java Naming and Directory Interface (JNDI) name.

About this task

For more information, see the topic Application bindings.

Before you do this task, it is assumed that the entity beans in your application are container-managed persistence (CMP) enterprise beans.

Note:
The EJB container handles the persistence of the bean attributes in the underlying persistent store. Specify which data store is used by binding an EJB module or individual EJB to a data source.

If you bind an EJB *module* to a data source, all beans in that module use the same data source for persistence. If you specify the data source at the bean level, then that data source is used instead.

See the assembly tool information center for the steps on how to complete this task.

Lightweight local operational mode for entity beans

WebSphere Application Server provides a special operational mode called *lightweight local* mode, which can improve the performance of entity bean methods. You can decide which entity beans in your application to run in this mode.

In lightweight local mode, the container streamlines the processing that it performs before and after every method on the local home interface and local business interface of the bean. This streamlining can result in improved performance when entity bean operations are called locally from within an application. Because some processing is skipped when running in lightweight local mode, this mode can be used in certain scenarios only.

Lightweight local mode is patterned somewhat after the Plain Old Java Object (POJO) entity model introduced in the Enterprise JavaBeans (EJB) 3.0 specification. Using lightweight local mode, you can obtain some of the performance advantages of the POJO entity model without having to convert your existing EJB 2.x application code to the new POJO model. You can apply lightweight local mode to both container-managed persistence (CMP) and bean-managed persistence (BMP) entity types that meet the specific criteria.

Attention: Entity beans are not supported in EJB 3.x modules.

When to use the lightweight local mode

Lightweight local mode is designed for entity beans that are created, found, and called using the *Session Facade* pattern. Under this pattern, entity bean local home and local business methods are called from within methods of a stateless session bean or stateful session bean. The session bean methods, which can be called remotely or locally, provide security control and transaction demarcation for the entity beans that are accessed by the session bean.

You can apply lightweight local mode only to an entity bean that meets the following criteria:
- The bean implements an EJB local interface.
- No security authorization is defined on the entity bean local home or local business interface methods.
- No *run-as* security attribute is defined on the local home or local business methods.
- The classes for the calling bean and the called entity bean are loaded by the same Java class loader.
- The entity bean methods do not call the WebSphere Application Server-specific Internationalization Service or Work Area Service.

The first criterion prevents CMP 1.x beans from supporting lightweight local mode, because the 1.x beans cannot have local interfaces.

In addition, lightweight local mode provides its fullest performance benefits only to entity bean methods that do not need to start a global transaction. This condition is true if you ensure that your entity bean also meets the following criteria:
- A global transaction is already in effect when the entity bean home or business method is called. Typically, this transaction is started by the calling session bean.
- The local business interface methods and the local home methods of the entity bean use the following transaction attributes only: REQUIRED, SUPPORTS, or MANDATORY.

If an entity bean method that is running in lightweight local mode must start a global transaction, the bean still functions normally but only a partial performance benefit is realized.

You can mark an entity bean that defines a remote interface or a TimedObject interface, in addition to the local interface, for lightweight local mode. However, the performance benefit is apparent only when the bean is called through its local interface.

Applying lightweight local mode to an entity bean

WebSphere Application Server provides a special operation mode called *lightweight local* mode, which can improve the performance of entity bean methods. You can decide which entity beans in your application to run in this mode.

About this task

You can apply lightweight local mode to specific EntityBean types within your application with the Marker interface technique.

Marker interface technique

About this task

Use the marker interface technique when a group of beans within the application is related through a common inheritance hierarchy, and all the beans in the hierarchy are to be marked. For an application with a large number of beans in a hierarchy, this technique is the most efficient.

To use a marker interface, code your bean implementation class to implement the **com.ibm.websphere.ejbcontainer.LightweightLocal** interface. The bean implementation class does not need to directly implement the interface; any parent class or interface can also implement it. For details, see the **com.ibm.websphere.ejbcontainer** package in the API section of the information center.

Developing read-only entity beans

In addition to the existing Enterprise JavaBeans (EJB) caching options, you can develop read-only entity beans.

About this task

You are most likely to want to use it under the following conditions:
- Your application uses data that change relatively infrequently. An example might be a retailing application that uses pricing data that only changes once a week or month.
- Your application can tolerate data that might be stale. The degree of *staleness* allowed by the EJB container is configurable by the user.
- The bean is coded in a thread-safe manner, so it can safely be invoked by multiple threads at once.

To use this function, declare the bean type as *read-only*. Declaring a bean type is done the same way that bean caching options are selected - through a selection list within an assembly tool.

To complete this task see the topic, Defining bean cache settings for a bean in the assembly tool information center.

Example: Using a read-only entity bean

A usage scenario and example for writing an Enterprise JavaBeans (EJB) application that uses a read-only entity bean.

Usage scenario

A customer has a database of catalog pricing and shipping rate information that is updated daily no later than 10:00 PM local time (22:00 in 24-hour format). They want to write an EJB application that has read-only access to this data. That is, this application never updates the pricing database. Updating is done through some other application.

Example

The customer's entity bean local interface might be:

```
public interface ItemCatalogData extends EJBLocalObject {

  public int getItemPrice();

  public int getShippingCost(int destinationCode);

}
```

The code in the stateless SessionBean method (assume it is a TxRequired) that invokes this EntityBean to figure out the total cost including shipping, would look like:

```
.....
// Some transactional steps occur prior to this point, such as removing the item from
 // inventory, etc.
 // Now obtain the price of this item and start to calculate the total cost to the purchaser

 ItemCatalogData theItemData =
    (ItemCatalogData) ItemCatalogDataHome.findByPrimaryKey(theCatalogNumber);

int totalcost = theItemData.getItemPrice();

// ...     some other processing, etc. in the interim
// ...
// ...

// Add the shipping costs
totalcost = totalcost + theItemData.getShippingCost(theDestinationPostalCode);
```

At application assembly time, the customer sets the EJB caching parameters for this bean as follows:

- ActivateAt = ONCE
- LoadAt = DAILY
- ReloadInterval = 2200

Note: The reloadInterval and reloadingEnabled attributes of the IBM deployment descriptor extensions, including both the WAR file extension (WEB-INF/ibm-web-ext.xmi) and the application extension (META-INF/ibm-application-ext.xmi) were deprecated.

On the first call to the getItemPrice() method after 22:00 each night, the EJB container reloads the pricing information from the database. If the clock strikes 22:00 between the call to getItemPrice() and getShippingCost(), the getShippingCost() method still returns the value it had prior to any changes to the database that might have occurred at 22:00, since the first method invocation in this transaction occurred prior to 22:00. Thus, the item price and shipping cost used remain in sync with each other.

Creating timers using the EJB timer service for enterprise beans

You can use enterprise beans to take advantage of the EJB timer service to schedule time-based events.

About this task

In support of the EJB 3.1 specification, you can create non-persistent EJB timers. This product also supports the expanded TimerService API for programmatic timer creation. In addition, you can configure the EJB container to automatically create a timer when the application starts.

WebSphere Application Server implements the Enterprise JavaBeans(EJB) timer service. Based on your business needs, you can use persistent timers or non-persistent timers. Persistent timers are helpful if you are creating a timer for a time-based event that requires assurance of timer existence beyond the life cycle of the server. Previously started persistent timers automatically start when your server starts and persist through server shutdowns and restarts. For example, you can use persistent timers to start a system application or send a status notification on the expiration of a timer. Non-persistent timers are helpful in non-critical situations where the timer actions are skipped or redone without negative business impacts, such as polling a temperature.

You can create timers programmatically. You can also create timers automatically by using the @Schedule annotation in the bean class, or by using the timer element in the ejb-jar.xml deployment descriptor. By automatically creating timers, you can schedule a timer without relying on your enterprise bean invocation to programmatically start an EJB timer service creation method.

Persistent timers

Persistent timers are implemented as a scheduler service task. By default, an internal or pre-configured scheduler instance is used to manage those tasks, and they persist in an Apache Derby database associated with the server process.

You can perform basic customizations to the internal scheduler instance. For information about customizing the scheduler instance, see the configuring a timer service information.

The creation and cancellation of Timers is transactional and persistent. If a Timer is created within a transaction and that transaction is later rolled back, the creation of the Timer is rolled back as well. Similar rules apply to the cancellation of a Timer. Previously started Timers are maintained across application server shutdowns and restarts.

Non-persistent timers

EJB 3.1 augments the EJB timer service to enable non-persistent EJB timers in addition to the persistent timers. Non-persistent timers have many of the same semantics and behavior as persistent timers, but without the overhead of a data store. Non-persistent timers have a different life cycle than persistent timers. Where persistent timers are maintained across application server shutdowns and restarts, non-persistent timers are active only while the application server is active. Unlike persistent timers, there are no commands to find or cancel non-persistent timers. Non-persistent timers are canceled when the application server is stopped or fails to remain in an active state.

As with persistent timers, the creation and cancellation of non-persistent timers is transactional. If a timer is created within a transaction and that transaction is later rolled back, the creation of the timer is rolled back as well. Similar rules apply to the cancellation of a timer.

You can configure non-persistent timers to share a thread pool with persistent timers, or to have a unique thread pool that is not shared with persistent timers.

Programmatically created timers

A programmatically created persistent timer is maintained across application server shutdowns and restarts, unless it is canceled. It is the responsibility of application code or system administrator to delete a programmatically created timer which is no longer wanted.

To programmatically create a timer that is associated with your enterprise bean, the bean calls the getTimerService() method on the applicable context instance to get a reference to the TimerService object. The bean also calls one of the TimerService methods, such as createTimer,

to specify the timer for the bean. This Timer instance is now associated with your bean. The TimerService methods are described in the EJB 3.1 specification. You can now pass the Timer instance to other Java code as a local object. After the Java code obtains the Timer instance, the code can use any of the methods defined by the javax.ejb.Timer interface, such as the cancel() or getTimeRemaining() methods.

In a clustered environment, a programmatically created persistent timer can run in any cluster member, but a programmatically created non-persistent timer runs only in the same JVM in which it was created.

Automatically created timers

The EJB 3.1 specification augments the EJB timer service to enable the automatic creation of a timer when your application starts without relying on a bean invocation to programmatically start one of the timer service timer creation methods. Use the @Schedule annotation or the timeout-method deployment descriptor element to automatically create timers. Automatically created timers are created by the container as a result of application deployment.

Note: The CancelEJBTimers command also cancels automatically created timers. When automatically created timers are canceled, the only way to recreate them is to uninstall the application and reinstall it again.

Timers in a clustered environment

In a clustered environment, a persistent timer runs only in one cluster member which might not necessarily be the same cluster member it was created in. A non-persistent timer runs in each cluster member that it was created in - automatic non-persistent timers run in each cluster member that contains the EJB.

Automatic persistent timers are removed from their persistent store when their containing module or application is uninstalled. Therefore, do not update applications that use automatic persistent timers with the Rollout Update feature. Doing so uninstalls and reinstalls the application while the cluster is still operational, which might cause failure in the following cases:

- If a timer running in another cluster member activates after the database entry is removed and before the database entry is recreated, then the timer fails. In this case, a com.ibm.websphere.scheduler.TaskPending exception is written to the First Failure Data Capture (FFDC), along with the SCHD0057W message, indicating that the task information in the database has been changed or canceled.
- If the timer activates on a cluster member that has not been updated after the timer data in the database has been updated, then the timer might fail or cause other failures if the new timer information is not compatible with the old application code still running in the cluster member.

When you use the proxy server in the product, do not define a scheduler at the cell level if that scheduler is configured as the one to use for the EJB timer service. Doing so prevents persistent timers from running. This can happen if the proxy server gets the scheduler lease. Since no applications run in the proxy server, there is no application code to handle the timer events that are sent by the scheduler.

Retries and missed timeouts

If you use EJB timers, you must understand the concepts of failure, retry, and missed timeouts.

- A failure is a timeout execution that is attempted, but does not succeed.
- A retry is an additional attempt to successfully execute a timeout that was previously attempted, but failed.
- A missed execution is a timeout that must have been attempted, but was not, because the server was unavailable or busy retrying a previously failed timeout.

The retry behavior reflects:

- Number of additional times that the server retries the failed timeout

- Interval between these server retries

The missed timeout behavior reflects:

- Whether the server eventually attempts missed timeouts, or not
- If missed timeouts are eventually attempted, when those attempts occur
- Interval between the missed timeout attempts

Table 48. Retry and missed timeout behavior. Retry and missed timeout behavior for both persistent and non-persistent timers.

Characteristic	Default behavior	Configurable
Number of retry attempts	As many as it takes to succeed. Persistent timers are temporarily deactivated if their scheduler failure threshold is reached in a server. For more information, see the topic about stopping tasks that are failing.	Yes, for non-persistent timers
Interval between retry attempts	First retry attempt is immediate. Subsequent retry attempts occur on the configured scheduler poll interval for persistent timers, and on the configured retry interval for non-persistent timers.	Yes
Missed timeout recovery	All missed timeouts are recovered.	No
When missed timeouts are recovered	Both persistent and non-persistent timers recover missed timeouts when the blocking retry attempts stop. Additionally, persistent timers recover timeouts when an unavailable server restarts.	No
Next timeout, after retry attempts are successful and missed timeouts are recovered	At the next originally scheduled time.	No

The configurable characteristics are configured on the scheduler instance for persistent timers and on the non-persistent timer configuration for non-persistent timers.

The following scenarios illustrate how the retry and missed timeout behavior influences the timeouts for both persistent and non-persistent timers.

Persistent timer scenario:

A persistent timer is created and configured to run for the first time at 10:00 am, and then run every hour after that. The scheduler supporting the timer is configured with a 30 second poll interval.

The timer runs at 10:00 am and fails because a database is unavailable. The timer is immediately retried, and retried again every 30 seconds when the scheduler is polled, and keeps failing, until 12:30 pm. At that moment, a retry attempt succeeds because the database is now back online, and therefore the server stops retrying the previously failed attempt.

Now, the server begins to work through the missed timeouts. First, it attempts the timeout that must have run at 11:00 am, which succeeds at 12:31 pm. When the scheduler is polled again 30 seconds later, it attempts the timeout that must have run at 12:00 pm, which succeeds at 12:32 pm. The server is now current, and the next timeout occurs at its originally scheduled time of 1:00 pm, as opposed to an hour after the last success, which would have been 1:32 pm. Going forward, the original schedule is maintained. The schedule is not updated based on the time of the last successful timeout.

Non-persistent timer scenario:

A non-persistent timer is created and configured to run for the first time at 10:00 am, and then run every hour after that. This non-persistent timer is configured to have a retry count of 5, and a retry interval of 30 minutes.

The timer runs at 10:00 am and fails because a database is not available. The timer is immediately retried, and fails again. Now, having executed the initial immediate retry, the server waits for the configured retry interval of 30 minutes between the next retry attempt. Retry attempts occur at 10:30 am and 11:00 am, and both fail. Finally, the fourth retry occurs at 11:30 am, and succeeds because the database is now back online.

Now, the server begins to work through the missed timeouts. The timeout that was originally scheduled for 11:00 am is immediately run, and succeeds at 11:31 am. The server is now current, and the next timeout occurs at its originally scheduled time of 12:00 pm (as opposed to an hour after the last success, which would be 12:32 pm).

Behavior of the getNextTimeout and getTimeRemaining methods

For both persistent and non-persistent timers, the Timer.getNextTimeout method returns a java.util.Date object that indicates the next time the timer is scheduled to execute. When you call the getNextTimeout method from a timeout callback method for an interval or calendar-based timer, the method returns the next scheduled time; for calendar-based timers with no future timeouts, the method throws the NoMoreTimeoutsException exception as required by the EJB 3.1 specification. When you call the getNextTimeout method from a timeout callback method for a single-action timer, the method returns the originally scheduled time. If a timeout callback method fails and retries are being attempted, the getNextTimeout method continues to return the originally scheduled time as if the failed execution had not occurred. In all cases, the Timer.getTimeRemaining method returns the difference in milliseconds between the getNextTimeout returned value and the current system time, which could result in a negative number, if the scheduled execution time was in the past.

Inheritance behavior of automatically created timers

Automatic timer methods in a hierarchy of bean classes cause multiple timers to be created. The number of timers associated with a timer method is not determined by the number of occurrences of the method in the source code. Instead, the number of timers associated with a timer method is determined by the number of beans with visibility to that method. For example:

```
@Stateless
public class Abean {

   @Schedule(hour="1")
   public void timerMethod1()


@Stateless
public class Bbean extends Abean {

   @Schedule(hour="2")
   public void timerMethod2()


@Stateless
public class Cbean extends Bbean {

   @Schedule(hour="2")
   public void timerMethod2()
```

In the previous bean class hierarchy, three automatic timers with callback method Abean.timerMethod1 are created, one for each bean instance with visibility to that method. One timer with callback method Bbean.timerMethod2 is created, and since that method is overridden by bean Cbean, only one timer with callback method Cbean.timerMethod2 is created.

In the previous example, when bean Abean is processed by the container, a single automatic timer is created, with callback method Abean.timerMethod1.

When the bean Bbean is processed by the container, an automatic timer is created with callback method Bbean.timerMethod2, and another automatic timer is created with callback method Abean.timerMethod1.

When the bean Cbean is processed by the container, an automatic timer is created with callback method CBean.timerMethod2. Another automatic timer is created with callback method Abean.timerMethod1. A timer for Bbean.timerMethod2 is not created when processing the bean Cbean. Bbean.timerMethod2 is not visible in the class hierarchy of Cbean because it is overridden by method Cbean.timerMethod2.

Consider another example like the previous one, if the @Stateless annotation were removed from classes Abean and Bbean, so that class Cbean is the only EJB. In that case, the only automatic timers created would be those visible to Cbean - one with callback method Abean.timerMethod1, one with callback method Cbean.timerMethod2.

Although beans can share identical code in an inherited bean callback method, the runtime behavior might be polymorphic. For example:

```
public class Employee {

   @Schedule(hour="1", dayOfMonth="-1", info = "payroll timer")
   public void getSalaryIncrease() {
      printChecks(salary * rate());
   }

   protected float rate() {
      return (float)1.01;
   }

}

public class Manager extends Employee {

   protected float rate() {
      return (float)1.15;
   }

}

public class Executive extends Manager {

   protected float rate() {
      return (float)1.30;
   }

}
```

In the previous example, each bean instance has an automatic timer with callback method getSalaryIncrease(). Although the same callback code is shared by each timer, note that the rate used in calculating the salary increase by each bean is different, due to polymorphism. That is, timer callback methods might be polymorphic in the same way that any Java methods might be.

Procedure

- To create timers programmatically, define the timeout method using the @Timeout annotation, the timeout-method deployment descriptor element, or by implementing the javax.ejb.TimedObject interface.

  A bean can have at most one timeout method for timers that are created programmatically. The timeout method is identified by the @Timeout annotation in the source code, or by the timeout-method deployment descriptor element contained in the ejb-jar.xml file. The timeout method must accept no parameters or a single parameter of type javax.ejb.Timer.

– Define the timeout method by using the @Timeout annotation to create timers programmatically. The following code is an example of using the @Timeout annotation within the MyBeanImpl class:

```
class MyBeanImpl implements MyBean{
@Timeout
void myTimeOutMethod()
```

– Define the timeout method by using the timeout method deployment descriptor element.

You can write your enterprise bean to implement the javax.ejb.TimedObject interface, including the ejbTimeout() method. If the bean implements the TimedObject interface, there is no need to annotate a timeout method, and it is not valid unless that method is the ejbTimeout() method.

The following code snippet illustrates using the timeout method deployment descriptor element within the `ejb-jar.xml` file:

```
<timeout-method>
<method-name>myTimeoutMethod</method-name>
<method-params>
<method-param>javax.ejb.Timer</method-param>
</method-params>
</timeout-method>
```

In this case, the container starts the myTimeoutMethod method on a bean instance when a timer for that instance has expired. The instance is notified by the myTimeoutMethod method of the defined time-based event so the instance can run the associated business logic, based on the timer expiration notification.

Now, you can pass the timer instance to other Java code as a local object. After the Java code obtains the timer instance, the code can use any of the methods defined by the javax.ejb.Timer interface, such as the cancel() or getTimeRemaining() methods.

Note: ForWebSphereApplication Server Version 6, no assembly tool supports the EJB timedObject object. To set the ejbTimeout method transaction attribute, you must manually enter the attributes in the deployment descriptor.

– Define the timeout method by using the javax.ejb.TimedObject interface.

- Write your enterprise bean to implement the javax.ejb.TimedObject interface, including the ejbTimeout() method. The bean calls the EJBContext.getTimerService() method to get an instance of the TimerService object. The bean calls the TimerService method to create a timer. This Timer is now associated with that bean.
- After you create the timer, you can pass the timer instance to other Java code as a local object.

The ejbTimeout method can contain any code that is typically placed in a business method of the bean. Method-level attributes such as *transaction* or *runAs* can be associated with this method by the application assembler. An instance of the timer that causes the method to run is passed in as an argument to ejbTimeout method. The following code snippet illustrates using the javax.ejb.TimedObject interface:

```
import javax.ejb.Timer;
import javax.ejb.TimedObject;
import javax.ejb.TimerService;

public class MyBean implements EntityBean, TimedObject {

  // This method is called by the container when the timer expires.
  public void ejbTimeout(Timer theTimer) {

    //You can place code that is typically placed in an EJB method.

     String whyWasICalled = (String) theTimer.getInfo():
     System.out.println("I was called because of"+ whyWasICalled);
  } // end of method ejbTimeout
```

A timer is created that starts the ejbTimeout method in 30 seconds. A simple string object is passed in at timer creation to identify the timer; for example:

```
// Instance variable to hold the EJB context.
private EntityContext theEJBContext;

// This method is called by the EJB container upon bean creation.
public void setEntityContext(EntityContext theContext) {

// Save the entity context passed in upon bean creation.
  theEJBContext = theContext;

}

// This business method causes the ejbTimeout method to begin in 30 seconds.
public void fireInThirtySeconds() throws EJBException  {

  TimerService theTimerService = theEJBContext.getTimerService();
  String aLabel = "30SecondTimeout";
  Timer theTimer = theTimerService.createTimer(30000, aLabel);

} // end of method fireInThirtySeconds

} // end of class MyBean
```

Note: The EJB 3.x programming model provides additional strategic ways to define persistent and non-persistent timers within your business environments. Although defining persistent timers using the ejbTimeout method with the TimedObject interface is still supported, take advantage of the easy-to-implement EJB annotations to create persistent and non-persistent timers to meet your business needs.

- Determine whether your timer must be single-action, interval, or calendar-based.
  - A single-action timer is scheduled to run once on a specific date or after a specified duration. To specify a single-action timer, use the TimerService.createSingleActionTime or TimerService.createTimer(Date, Serializable) APIs.
  - An interval timer is scheduled to begin on a specific date or after a specified duration and continues to run at a fixed rate until it is canceled. To specify an interval timer, use the TimerService.createIntervalTimer, TimerService.createTimer(long, long, Serializable), or TimerService.createTimer(Date, long, Serializable) APIs.
  - A calendar-based timer is scheduled to run at specific dates and times. To specify a calendar-based timer, either use an automatically created timer or use the TimerService.createCalendarTimer APIs, which accept a javax.ejb.ScheduleExpression argument.

    Note: When the EJB container stores a serialized calendar-based timer in the database, the serialized format is incompatible with the format used in WebSphere Application Server Version 7 and lower. If you use calendar-based timers, you must not configure the scheduler for the EJB container to use database tables that are shared with an older version of WebSphere Application Server.

    When using the TimerService.createCalendarTimer API, several attributes are specified to form an expression like the *cron* job scheduling daemon. The first set of attributes is used to control how date calculations are performed:

Table 49. Attributes that control how dates are calculated.. Attributes that control how dates are calculated

Attribute	Description
start	Specifies a date value that is the inclusive starting point for calculating timeouts. The default value is null, which means the timer can start at any time.
end	Specifies a date value that is the inclusive ending point for calculating timeouts. The default value is null, which means the timer continues to run indefinitely.

Table 49. Attributes that control how dates are calculated. (continued). Attributes that control how dates are calculated

Attribute	Description
timezone	Specifies a valid time zone according to the java.util.TimeZone API. The default value is null, which corresponds to the default time zone of the host server.

The second set of attributes determines when the timer runs. All attributes have a set of allowable values in addition to a special wild-card value *, which represents all possible values for that attribute.

Table 50. Attributes that determine when the timer runs.. Attributes that determine when the timer runs

Attribute	Description
second	Specifies an integer in the range 0 to 59; default 0.
minute	Specifies an integer in the range 0 to 59; default 0.
hour	Specifies an integer in the range 0 to 23; default 0.
dayOfMonth	Specifies an integer in the range 1 to 31; default *. Alternatively, the value can be: • The keyword **Last**, which corresponds to the last day of the month. • An integer in the range -1 to -7, which corresponds to a day before the last day of the month. For example, in a month with 31 days, the value -3 would correspond to the 28 day of the month. • The keyword **1st**, **2nd**, **3rd**, **4th**, **5th**, or **Last** followed by a day of the week keyword **Sun**, **Mon**, **Tue**, **Wed**, **Thu**, **Fri**, or **Sat**. For example, you might use **3rd Sun** and **Last Wed**. You must use the abbreviated English ordinals and keywords.
dayOfWeek	Specifies an integer in the range 0 to 7; default *. These values correspond to the keywords **Sun**, **Mon**, **Tue**, **Wed**, **Thu**, **Fri**, and **Sat**, which can be used instead. The values 0 and 7 both correspond to the keyword **Sun**. When both dayOfMonth and dayOfWeek are a value other than *, only one of the attributes needs to match a given day.
month	Specifies an integer in the range 1 to 12; default *. These values correspond to the keywords **Jan**, **Feb**, **Mar**, **Apr**, **May**, **Jun**, **Jul**, **Aug**, **Sep**, **Oct**, **Nov**, and **Dec**, which can be used instead.
year	Specifies any four-digit calendar year; default *.

You can specify each of these attributes using an **x-y** range notation. For example, **0-3** is a valid range for the hour attribute and **Jun-Aug** is a valid range for the month attribute. Additionally, the second, minute, and hour attributes can be specified using an **x/y** increment notation. For example, **30/10** is a valid increment for the minute attribute, which includes the values 30, 40, and 50.

All attributes can also be specified as comma-delimited lists of single values or ranges of values. For example, the value **4,10-12** is valid for the hour attribute, and it corresponds to the times 4 AM, 10 AM, 11 AM, and 12 PM.

The following example uses the America/New_York time zone value for its calendar operations. The timer runs on the last Friday of the months January, February, March, and June. On those days, it runs every two hours starting at 1:30 AM:

```
TimerService theTimerService = theEJBContext.getTimerService();
Timer theTimer = theTimerService.createCalendarTimer(new ScheduleExpression()
 .timezone("America/New_York")
 .month("Jan-Mar, Jun")
 .dayOfMonth("Last Fri")
 .hour("1/2")
 .minute(30))
```

- To create timers automatically, define the timeout callback method using the @Schedule or @Schedules annotation or using the timer deployment descriptor element. Use annotations to write metadata for EJBs inside your source code. For more details, see the information about EJB 3.x metadata annotations.

A bean can have multiple timeout callback methods for timers that are created automatically. The timeout callback method and automatic timers are identified by the @Schedule or @Schedules annotation in the source code or by using the timer deployment descriptor element in the ejb-jar.xml file. If you specify both annotations for the same method or if you specify an annotation and the timer deployment descriptor element for the same method, the metadata is combined and multiple automatic timers are created. The timeout callback methods must have a void return type and must accept either no parameters or a single parameter of type javax.ejb.Timer. All automatic timers are calendar-based. See information about scheduling calendar-based timers for additional information.

The server creates automatic timers when the application starts for the first time. Subsequent application start operations detect that timers have already been created for this application, and the timers are not recreated even if they were subsequently canceled. For persistent automatic timers, the server stores this data in the scheduler tables. If your application is installed on a server cluster, then you must configure the EJB container on each server to use a shared cluster-scoped scheduler. If your application is installed on independent servers, then you must ensure that each server is configured with a unique database or unique database table prefix. See information about clustered environment considerations for timer service.

The application server automatically removes persistent automatic timers from the database when you uninstall the application while the server is running. If the application server is not running, you must manually delete the automatic timers from the database. Additionally, if you add, remove, or change the metadata for automatic timers while the server is not running, you must manually delete the automatic timers.

– Define the automatic timer by using the @Schedule annotation. The following example uses @Schedule annotation:

```
class MyBeanImpl implements MyBean {
    @Schedule(hour="20", info="single timer", persistent=false)
    public void automatic(Timer t) {
        // ...
    }
}
```

The @Schedule annotation has second, minute, hour, dayOfMonth, dayOfWeek, month, year, and timezone elements that correspond to the same attributes of javax.ejb.ScheduleExpression. The annotation also has a persistent element that you can use to specify whether the server uses the EJB timer service scheduler to persist the timer. By default, automatic timers are persistent. Finally, the annotation also has an info element that you can use to specify application information that is delivered as a java.lang.String object with the javax.ejb.Timer object when the timeout callback method runs.

– Define multiple automatic timers for the same method by using the @Schedules annotation. The following example uses the @Schedules annotation:

```
class MyBeanImpl implements MyBean {
    @Schedules(
        @Schedule(hour="1" info="1AM timer", persistent=false),
        @Schedule(minute="0/30" info="30 minute timer")
    )
    public void automaticMultiple(Timer t) {
        // ...
    }
}
```

– Define timers using the deployment descriptor element. You can use the following subelements of the timer element:

Table 51. Subelements of the timer element. Subelements of the timer element

Element	Required or Optional	Description
schedule	Required	Includes optional second, minute, hour, day-of-month, day-of-week, month, and year subelements that correspond to the same attributes of javax.ejb.ScheduleExpression.

Table 51. Subelements of the timer element (continued). Subelements of the timer element

Element	Required or Optional	Description
start and end	Optional	Specify the inclusive starting and ending points for calculating timeouts.
timeout-method	Required	Specifies the timeout callback method.
persistent	Optional	Specifies whether the server uses the EJB timer service scheduler to persist the timer. By default, automatic timers are persistent.
timezone	Optional	Corresponds to the same attribute of javax.ejb.ScheduleExpression.
info	Optional	Specifies application information that is delivered as a java.lang.String object with the javax.ejb.Timer object when the timeout callback method runs.

The following example uses the timer deployment descriptor element:

```
<session>
  <ejb-name>MyBeanImpl</ejb-name>
  <timer>
    <schedule>
      <hour>20</hour>
    </schedule>
    <start>2000-01-01T13:00:00</start>
    <timeout-method>
      <method-name>automatic</method-name>
    </timeout-method>
    <persistent>false</persistent>
    <timezone>America/New_York</timezone>
    <info>single timer</info>
  </timer>
</session>
```

- Determine whether your timer is persistent or non-persistent.
  - Create a persistent timer.
  - Create a non-persistent timer.
- Deploy your EJB application. After you deploy your EJB application, the enterprise bean must run so that the createTimer methods are called before the timer is created programmatically. If the timer is automatically created, the timer starts when the EJB application is started.

Results

You have programmatically or automatically configured an EJB timer that is either persistent or non-persistent.

Clustered environment considerations for timer service

In a single server environment, it is clear which server instance should invoke the timeout method of the bean on a given bean. In a multi-server clustered environment there are other considerations governing the behavior.

WebSphere Application Server implements the Enterprise JavaBeans (EJB) Timer Service. Based on your business needs, you can use persistent timers or non-persistent timers. Persistent timers are helpful if you are creating a timer for a time-based event that requires assurance of timer existence beyond the life cycle of the server to survive server shutdowns and restarts. Previously started persistent timers automatically start when your server starts and they require a database instance. Non-persistent timers do not use a data store and are canceled when the application server is stopped or fails to remain in an active state. Non-persistent timers exist only on the server where they are created. In a clustered environment, if your EJB application automatically creates a non-persistent timer and this application is mirrored on multiple servers, each server has its own non-persistent timer that runs within that server environment. A programmatically created non-persistent timer only runs in the cluster member that it was created in.

When configuring a persistent timer in a multi-server clustered server environment, consider the following possibilities for the server instance to invoke the timeout method on a given bean:

- Separate timer service database per server process or cluster member. This is the default configuration. Only the server instance or cluster member that created the Timer can access the Timer and run the timeout method of the bean. If the server instance is unavailable, the Timer does not run at the specified time, and does not run until the server is restarted. Also, if an enterprise bean calls the *getTimers()* method, only those timers created on the server instance are found. This can cause unexpected behavior if the enterprise bean attempts to cancel all timers associated with it; for example, when the enterprise bean is removed. This configuration is NOT recommended for production level systems.
- Shared or common timer service database for the cluster. Timers can be created and accessed on any server process or cluster member. Timers created in one server process are found by the *getTimers()* method on other server processes in the cluster. When an entity bean is removed, all timers, no matter where created, are cancelled. However, all timers are executed on a single server in the cluster, that is, the timeout method of the bean is run for all timers on a single server. Which server executes the timers varies depending on which server process obtains a lock on the common database tables. If the server executing timers becomes unavailable, then another server or cluster member takes over and begins executing all timers at their scheduled time. This is the recommended configuration for all production level systems.
-

  Note: When using the EJB Timer service in an application using multi-threaded database access, application flow can introduce deadlock problems.
  To avoid this, use the wsPessimisticUpdate access intent. This access intent causes the finder method in your application to run a *select for update* statement instead of a generic select. This in turn prevents the lock escalation deadlock when multiple threads try to escalate their locks to perform an update.

See the information on using the EJB timer service for enterprise beans to learn how to configure the data source (database) to be used for each server process timer service.

Note: Once the data source for the timer service is changed to point to a different database, the server process automatically attempts to create the required tables in that database on the next server start.
If the user ID associated with the start of the server process is not authorized to create database tables in the configured timer service database, then the tables must be created manually.

Note: When you use the proxy server in the product, do not define a scheduler at the cell level if that scheduler is configured as the one to use for the EJB timer service. Doing so prevents persistent timers from running. This can happen if the proxy server gets the scheduler lease. Since no applications run in the proxy server, there is no application code to handle the timer events that are sent by the scheduler.

Timer service commands

Information about Enterprise JavaBeans (EJB) timers is specific to the application that the timers are created for, and the timers are not visible outside of that application. Therefore, when you manage EJB timers, use the application that contains the enterprise bean and creates the EJB timer.

You can use the following commands during application development to provide basic EJB timer management functions. These commands are not available on *client only* installations.

findEJBTimers

This command displays information about existing persistent EJB timers, based on specified filter criteria.

This command displays information about existing persistent EJB timers, based on specified filter criteria. The syntax for this command is:

```
findEJBTimers server filter [options]
   filter: -all | -timer | -app [-mod [-bean ]]
           -all
           -timer timer id
           -app   application name
           -mod   module name
           -bean  bean name

  options: -host  host name
           -port  portnumber
           -conntype connector type
           -user  userid
           -password password
           -quiet
           -logfile filename
           -replacelog
           -trace
           -help
```

The following options exist:

server Specifies the name of the server process where the EJB timers are located

-all Specifies to find all EJB timers associated with the server process

timer id
Specifies the EJB Timer ID that uniquely identifies the timer

application name
Specifies to find all EJB timers associated with the application

module name
Specifies to find all EJB timers associated with the module

bean name
Specifies to find all EJB timers associated with the enterprise bean

host name
Specifies the host name of the server process

portnumber
Specifies the port of the server process

connector type
Specifies the type of connection. For example, SOAP, RMI, or NONE.

userid Specifies the user to use when connecting to the server process

password
Specifies the password to use when connecting to the server process

quiet Specifies to disable output

logfile Specifies to direct output to a file

replacelog
Specifies to clear the existing log before executing the command

trace Specifies to enable trace

help Specifies to provide command-specific help

Note: If the server you specify is configured to use a scheduler instance that is shared by multiple servers, then EJB timers that are created in any of the server processes might be found.

See the information about locating EJB timers using the findEJBTimers command.

cancelEJBTimers

This command cancels and removes from persistent storage EJB persistent timers based on the specified filter criteria.

The syntax for this command is:

```
cancelEJBTimers server filter [options]
   filter: -all | -timer | -app [-mod [-bean ]]
           -all
           -timer timer id
           -app   application name
           -mod   module name
           -bean  bean name

  options: -host  host name
           -port  portnumber
           -conntype connector type
           -user  userid
           -password password
           -quiet
           -logfile filename
           -replacelog
           -trace
           -help
```

the following options exist:

server Specifies the name of the server process where the EJB timers are located

-all Specifies to find all EJB timers associated with the server process

timer id
Specifies the EJB Timer ID that uniquely identifies the timer

application name
Specifies to find all EJB timers associated with the application

module name
Specifies to find all EJB timers associated with the module

bean name
Specifies to find all EJB timers associated with the enterprise bean

host name
Specifies the host name of the server process

portnumber
Specifies the port of the server process

connector type
Specifies the type of connection. For example, SOAP, RMI, or NONE.

userid Specifies the user to use when connecting to the server process

password
Specifies the password to use when connecting to the server process

quiet Specifies to disable output

logfile Specifies to direct output to a file

replacelog
Specifies to clear the existing log before executing the command

trace Specifies to enable trace

help Specifies to provide command-specific help

Note: If the server you specify is configured to use a scheduler instance that is shared by multiple servers, then EJB timers that are created in any of the server processes might be canceled.

For an example of the cancelEJBTimers command, see the topic CancelEJBTimers command example.

findEJBTimers command:

The following examples illustrate how to use the **findEJBTimers** command to find Enterprise JavaBeans (EJB) timers and explain the output statement. For relevant parameters and syntax information, read about the timer service commands.

To find all EJB timers on a server called `server1`, enter the following command in the `<install-root>\profiles\<profile>\bin` directory:

```
findEJBTimers server1 -all
```

To find all EJB timers on `server1`, associated with the `Increment` bean in the **DefaultApplication**, enter the following command in the `<install-root>\profiles\<profile>\bin` directory:

```
findEJBTimers server1 -app DefaultApplication.ear
                      -mod Increment.jar -bean Increment
```

When EJB timers matching the filter criteria are found, the output appears similar to the following:

```
EJB Timer : 252     Expiration: 5/25/10 10:53 AM     Single
   EJB    : TimerPtestApp, TimerPtestEJB.jar, NoMoreTimeoutsBean
   Info   : Single
   Programmatic timer
EJB Timer : 253     Expiration: 5/25/10 11:47 AM     Calendar
   EJB    : TimerPtestApp, TimerPtestEJB.jar, NoMoreTimeoutsBean
   Automatic timer
Calendar expression: [start=null, end=null, timezone=null, seconds="52",
           minutes="47", hours="11", dayOfMonth="25", month="5", dayOfWeek="*",
           year="2010"]
2 EJB Timer tasks found
```

In this output, the following elements exist:

- `EJB Timer` is the unique identifier of the timer.
- `Expiration` is the next time the timer is expected to execute.
- `Calendar expression` is the calendar expression that defines the timer expiration intervals and frequencies, if the timer expirations were defined by a ScheduleExpression. All automatic timers have expirations defined by a ScheduleExpression.
- `EJB Key` is the `toString()` method output of the primary key for and entity bean. For other EJB types, `Not Available` will be output.
- `Info` is the `toString()` method output of the object passed by the application when the EJB timer was created.

Only the first forty bytes of `toString()` output are displayed for the primary key and timer information. This information is only useful if the application overrides the `toString()` method for these objects.

CancelEJBTimers command example:

The following examples illustrate how to use the command to cancel Enterprise JavaBeans (EJB) timers.

To use the cancelEJBTimer command to cancel all EJB timers on a server called **server1**:

```
cancelEJBTimers server1 -all
```

To cancel all EJB timers on **server1**, associated with the *Increment* bean in the **DefaultApplication**:

```
cancelEJBTimers server1 -app DefaultApplication.ear -mod Increment.jar -bean Increment
```

To cancel a specific EJB timer identified through the FindEJBTimers command or from a system log entry indicating a problem or failure:

```
cancelEJBTimers server1 -id 25
```

Increment in the *DefaultApplication* does not implement the *TimedObject* interface, and so could not actually have associated EJB Timers. Increment is used merely for illustrative purposes in this example.

Note: The CancelEJBTimers command also cancels automatically created persistent timers. When automatically created persistent timers are cancelled, the only way to recreate them is to uninstall the application and reinstall it again.

EJB command group:

The EJB command group for the AdminTask object provides commands that you can use to manipulate enterprise beans.

removeAutomaticEJBTimers

Applications or modules use annotations or XML to instruct the application server to automatically create EJB timers.

Automatically created timers are persisted in the scheduler instance associated with the server that the application or module is running on at the moment the automatic EJB timer is created. Schedules are configured on a per server basis, and thus it is possible for each server in your topology to use a unique scheduler instance. In this case, the scheduler instance specific to each server supports the EJB timers running in that server.

Each scheduler instance is associated with a set of database tables. If you have multiple scheduler instances, configure each scheduler instance with a unique prefix to ensure the instance maps to a unique set of database tables.

When the application or module that requested the automatic EJB timers are removed from a server, the automatic EJB timers must be removed from the corresponding scheduler instance. If the application or module was installed on multiple servers, and each of those servers used a unique scheduler instance, then the timers must be removed from each of those scheduler instances. In other words, automatically created EJB timers are removed on a per server basis.

In some cases, the removal or update of the application or module results in the removal of the automatically created EJB timers from the scheduler instance. In this scenario, no user action is required.

However, in other cases, the removal, or update of the application or module does not result in the removal of the automatically created EJB timers from the scheduler instance. In this case, you must manually remove the EJB timers using the **removeAutomaticEJBTimers** command.

The command is only supported in a connected mode. In a network deployment topology, the deployment manager, NodeAgent, and the managed server that contains the scheduler instance must all be running. In a base topology, the stand-alone server must be running.

In a Rational Application Developer loose configuration scenario, you must manually remove your automatically created EJB timers. Also, programmatically created EJB timers, which are not the same as automatically created EJB timers, are not automatically removed or removed by this command.

In a network deployment topology, when a single module is only installed on a subset of the servers in the topology, the automatically created timers associated with that module must be removed from the scheduler instance(s) associated with that subset of servers only. A scheduler instance corresponding to a server that does not have the module installed on it does not need to be cleaned up.

Target Object: None

Required parameters:

-appName
The name of the application that requested the automatically created EJB timers you want to remove. (String, required)

-serverName
The name of the server that runs the application or module that contains the automatically created EJB timers you want to remove. This parameter represents the logical name of an application server, not a host name. (String, required)

Optional parameters:

-schedulerJNDIName
This parameter represents the JNDI name of the scheduler instance that persists the automatically created EJB timers you want to remove.

A server instance is always configured to use a particular scheduler instance to support automatically created EJB timers. You can explicitly configure which scheduler instance is used, or you can choose to not explicitly configure a scheduler instance. In the latter case a default scheduler instance is used.

If the scheduler instance that contains the automatically created EJB timers you want to remove is the same scheduler instance that is currently configured for the server, then you can omit this parameter. In this case, the command examines the configuration, discovers the scheduler instance that is currently configured, and uses that.

However, if the scheduler instance that is currently configured is *not* the one that contains the EJB timers you want to remove, then specify the JNDI name of the scheduler instance that does contain these timers. (String, optional)

-nodeName
The name of the node that contains the server. (String, required)

-moduleName
The name of the module that requested the automatically created EJB timers you want to remove. If you want to remove all automatically created timers in the application, regardless of which module they are defined in, then this parameter is omitted. This parameter is only specified when you want to remove the automatically created timers requested by one module in the application, but not the timers requested by another module in the same application. (String, optional)

Return value: None

The following information helps to determine when this command is needed:

- WebSphere Application Server attempts to remove automatically created EJB timers when all the following conditions are met:
  - One of the following actions is performed:
    - Application uninstallation
    - Application update
    - Module uninstallation
    - Module update
  - The action was performed in a connected mode (not **wsadmin -conntype none**).

  - The needed servers were running at the time of the action.
    - For a network deployment topology, the deployment manager, node agent, and managed server containing the automatically created EJB timers were all running.
    - For a base topology, the stand-alone server was running.
  - The correct follow-up action was performed.
    - For a network deployment topology, you save your changes and synchronize them to the node agent. If you save your changes but do not synchronize them, then the automatically created EJB timers are *not* removed. The automatically created EJB timers are only removed during the sync processing.
    - For a base topology, you save your changes.
  - The database supporting the scheduler instance was running.
- WebSphere Application Server does *not* remove automatically created EJB timers when *any* of the following conditions occur:
  - Any action is performed other than an update or uninstallation of an application or module.
  - The action (even if it is an application/module uninstallation/update) was performed in disconnected mode.
  - The needed servers were not running.
    - For a network deployment topology, the deployment manager, node agent, or managed server was not running.
    - For a base topology, the stand-alone server was not running
  - The correct follow-up action was not performed.
    - For a network deployment topology, either the save or synchronize was not performed.
    - For a base topology, the save was not performed.
  - The database supporting the scheduler instance was not running.

If WebSphere Application Server encounters an error when attempting to remove automatically created EJB timers from a server, then a warning is written to log file.

If an error occurs, or if the application server did not attempt to remove automatically created EJB timers, or if you are not sure if the automatically created EJB timers were removed, manually issue the **removeAutomaticEJBTimers** command to ensure that the automatically created EJB timers are removed. If the automatically created timers were actually removed from the scheduler instance, running the command is unnecessary, but does no harm.

If you are running in a clustered environment, and your cluster contains multiple nodes, and each of those nodes contains a server that maps to the same cluster level scheduler instance, then the automatically created timers only must be removed from one of the servers. This is because the shared scheduler instance is updated, and all servers using that shared scheduler instance see the change.

As a result, if a server in one node is not running and you receive a log warning that the automatic timers could not be removed from it, but you know that server shares a cluster-level scheduler instance with a server in a different node that was successfully cleared, then no user action is required because the shared scheduler instance has already been updated.

The same is true when there are multiple servers in a cluster, and they are all part of the same node, and share a single cluster-level scheduler instance, and one or more of those cluster members are not running. In this case, the application server issues a log warning that the automatic timers could not be removed from those servers. However if you know that they share a common scheduler instance, and one of the cluster members was successfully cleared, then no user action is required because the shared scheduler instance has already been updated.

If you are running in a network deployment topology and have multiple servers, the type of scheduler used (default versus custom configured) has an impact on performance in regards to scheduler cleanup. The default EJBContainer scheduler is unique per server. If you are using the default EJBContainer scheduler instance, and you have five servers, this means you have five unique scheduler instances, and the automatic timers must be removed from all five of them when the application is updated or removed. However, if you are using a single shared, custom configured scheduler instance, then the automatic timers must only be removed once, from that one scheduler instance.

Example 1

Topology:

- Deployment manager running on workstation A.
- NodeAgent running on workstation B. The node is called Node01.
- Managed server server1 running on workstation C.
- Managed server server2 running on workstation D.
- Security is enabled.
- server2 is currently configured to use the same scheduler instance that contains the timers.

Background:

The testApp application was uninstalled in a connected mode from the administrative console. You want to remove all the automatically created timers in the application, regardless of which module requested them.

The deployment manager, node agent, and server1 servers were running, and the automatic EJB timers were removed from server1. However, server2 was not running, and so the automatic EJB timers were not removed from server2.

Now, you must manually remove the automatic EJB timers from server2.

Action:

- Using Jacl, from a connected wsadmin session in the deployment manager enter the following line of code:

  ```
  $AdminTask removeAutomaticEJBTimers "-appName testApp -serverName server2 -nodeName Node01"
  ```

- Using Jython, from a connected wsadmin session in the deployment manager enter the following line of code:

  ```
  AdminTask.removeAutomaticEJBTimers('-appName testApp -serverName server2 -nodeName Node01')
  ```

Example 2

Topology:

- Stand-alone server1 running on workstation A. The node is called Node01.
- Administrative security is disabled.
- server1 was configured to use scheduler instance `jndi/sched_1` when the automatic EJB timers were created. However, server1 is currently configured to use scheduler instance `jndi/sched_2`.

Background:

The module, mod1, from application, testApp, was uninstalled, but because server1 was configured to use the `jndi/sched_2` instance at the moment of uninstallation, the automatic EJB timers were not removed from scheduler instance `jndi/sched_1`.

Now, you must manually remove the automatic EJB timers from the `jndi/sched_1` scheduler instance on server1.

The application contains modules, mod1 and mod2. Both of these modules requested automatically created EJB timers. The mod2 module is still installed, and you still need the automatically created EJB timers it requested. You only want to remove the automatically created EJB timers requested by mod1.

Action:

- Using Jacl, from a connected wsadmin session in the stand-alone server enter the following line of code:

```
$AdminTask removeAutomaticEJBTimers "-appName testApp -moduleName mod1 -serverName server1 -nodeName Node01 -schedulerJNDIName jndi/sched_1"
```

- Using Jython, from a connected wsadmin session in the stand-alone server enter the following line of code:

```
AdminTask.removeAutomaticEJBTimers('-appName testApp -moduleName mod1 -serverName server1 -nodeName Node01 -schedulerJNDIName jndi/sched_1')
```

Example: Using the Timer Service with the TimedObject interface:

This example shows the implementation of the ejbTimeout() method that is called when the scheduled event occurs.

Note: The EJB 3.x programming model provides additional strategic ways to define persistent and non-persistent timers within your business environments. Although defining persistent timers using the ejbTimeout method with the TimedObject interface is still supported, take advantage of the easy-to-implement EJB annotations to create persistent and non-persistent timers to meet your business needs. See the using the EJB timer service for enterprise beans information to learn more about creating persistent and non-persistent timers using annotations or defining your own timeout methods within a deployment descriptor.

The ejbTimeout method can contain any code that is typically placed in a business method of the bean. Method-level attributes such as transaction or runAs can be associated with this method by the application assembler. An instance of the Timer object that causes the method to fire is passed in as an argument to the ejbTimeout method.

```
import javax.ejb.Timer;
import javax.ejb.TimedObject;
import javax.ejb.TimerService;

public class MyBean implements EntityBean, TimedObject {

  // This method is called by the EJB container upon Timer expiration.
  public void ejbTimeout(Timer theTimer) {

    // Any code typically placed in an EJB method can be placed here.

     String whyWasICalled = (String) theTimer.getInfo():
     System.out.println("I was called because of"+ whyWasICalled);
  } // end of method ejbTimeout
```

A Timer is created that starts the ejbTimeout method in 30 seconds. A simple string object is passed in at Timer creation to identify the Timer.

```
// Instance variable to hold the EJB context.
private EntityContext theEJBContext;

// This method is called by the EJB container upon bean creation.
public void setEntityContext(EntityContext theContext) {

// Save the entity context passed in upon bean creation.
  theEJBContext = theContext;

}

// This business method causes the ejbTimeout method to begin in 30 seconds.
public void fireInThirtySeconds() throws EJBException  {
```

```
  TimerService theTimerService = theEJBContext.getTimerService();
  String aLabel = "30SecondTimeout";
  Timer theTimer = theTimerService.createTimer(30000, aLabel);

} // end of method fireInThirtySeconds

} // end of class MyBean
```

Developing enterprise beans

One of two enterprise bean development scenarios is typically used with the product. The first is command-line using Ant, Make, Maven or similar tools. The second is an IDE-based development and build environment. The steps in this article focus on development without an IDE.

Before you begin

Enterprise JavaBeans (EJB) 2.x beans only: Design a J2EE application and the enterprise beans that it needs.

- Before developing entity beans with container-managed persistence (CMP), read the topic Concurrency control.

EJB 3.x beans only: Design a Java EE application and the enterprise beans that it needs.

- Before developing entity beans with CMP, read the topic, Concurrency control. Keep in mind that EJB 3.x modules do not support entity beans. You must continue to place entity beans in your EJB 2.x-level modules.

About this task

The two basic approaches to select tools for developing enterprise beans are as follows:

- You can use one of the available IDE tools that automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. The Rational Application Developer product is the recommended IDE.

  Add install_root/dev/JavaEE/j2ee.jar to the IDE project build path to resolve compilation dependencies on the new EJB 3.x API classes. Code assist works when this JAR file is added to the project build path. If you define a server (see J2EE Perspective), point the server to the product installation directory. When you create a Java EE related project in Rational Application Developer, the project automatically adds install_root/dev/JavaEE/j2ee.jar. to the project build path.
- If you have decided to develop enterprise beans without an IDE, you need at least an ASCII text editor. You can also use a Java development tool that does not support enterprise bean development. You can then use tools available in the Java Software Development Kit (SDK) and in this product to assemble, test, and deploy the beans.

  Like the assembly tool, a standard Java EE command-line build environment requires some change to use the EJB 3.x modules. As with previous Java EE application development patterns, you must include the `j2ee.jar` file located in the install_root/dev/JavaEE directory on the compiler class path. An example of a command-line build environment using Ant is located in the install_root/samples/src/TechSamp directory.

The following steps primarily support the second approach, development without an IDE.

Procedure

1. If necessary, migrate any pre-existing code to the required version of the EJB specification.

   Applications written to the EJB specification versions 1.1, 2.0, and 2.1 can run unchanged in the EJB 3.x container. See the topic Migrating enterprise bean code to the supported specification.
2. Write and compile the components of the enterprise bean.
   - At a minimum, a session bean developed with the EJB 3.x specifications requires a bean class.

- At a minimum, an EJB 1.1 session bean requires a bean class, a home interface, and a remote interface. An EJB 1.1 entity bean requires a bean class, a primary-key class, a home interface, and a remote interface.
- At a minimum, an EJB 2.x session bean requires a bean class, a home or local home interface, and a remote or local interface. An EJB 2.x entity bean requires a bean class, a primary-key class, a remote home or local home interface, and a remote or local interface. The types of interfaces go together: If you implement a local interface, you must also define a local home interface.

  Attention: The primary-key class can be *unknown*. See the topic Unknown primary-key class for more information.
- A message-driven bean requires only a bean class.

3. For each entity bean, complete work to handle persistence operations.

   For EJB 3.x modules, consider using the Java Persistence API (JPA) specification to develop plain old Java Object (POJO) persistent entities. Review the topic Java Persistence API for more information. If you choose to develop entity beans to earlier EJB specifications, follow these steps:
   - Create a database schema for the entity bean persistent data.
     – For entity beans with CMP, you must store the bean persistent data in one of the supported databases. The assembly tool automatically generates SQL code for creating database tables for CMP entity beans. If your CMP beans require complex database mappings, it is recommended that you use Rational Application Developer to generate code for the database tables. For more information about using the assembly tools, see the assembly tool information center.
     – For entity beans with bean-managed persistence (BMP), you can create the database and database table by using the database tools or use an existing database and database table.

     For more information about creating databases and database tables, review your database documentation.
   - **(CMP entity beans for EJB 2.x only)**

     Define finder queries with EJB Query Language (EJB QL).

     With EJB QL, you define finders in terms of CMP fields and container-managed relationships, as follows:
     – *Public* finders are visible in the bean home interface. Implemented in the bean class, they return only remote interfaces and collection types.
     – *Private* finders, expressed as SELECT statements, are used only within the bean class. They can return both local and remote interfaces, dependent values, other CMP field types, and collection types.
   - **(CMP entity beans for EJB 1.1 only: an IBM extension)** Create a finder helper interface for each CMP entity bean that contains specialized finder methods (other than the findByPrimaryKey method).

     Logic other than the findByPrimaryKey method is required for each finder method that is contained in the home interface of an entity bean with CMP:
     – The logic must be defined in a public interface named *Name*`BeanFinderHelper`, where *Name* is the name of the enterprise bean, for example, AccountBeanFinderHelper.
     – The logic must be contained in a String constant named *`findMethodName`* `WhereClause`, where *findMethodName* is the name of the finder method. The String constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method arguments when that method is called.

Example: Using a read-only entity bean

This usage scenario and example shows how to write an Enterprise JavaBeans (EJB) application that uses a read-only entity bean.

Usage scenario

A customer has a database of catalog pricing and shipping rate information that is updated daily no later than 10:00 PM local time (22:00 in 24-hour format). They want to write an EJB application that has read-only access to this data. That is, this application never updates the pricing database. Updating is done through some other application.

Example

The customer's entity bean local interface might be:

```
public interface ItemCatalogData extends EJBLocalObject {

  public int getItemPrice();

  public int getShippingCost(int destinationCode);

}
```

The code in the stateless SessionBean method (assume it is a TxRequired) that invokes this EntityBean to figure out the total cost including shipping, would look like:

```
.....
// Some transactional steps occur prior to this point, such as removing the item from
 // inventory, etc.
 // Now obtain the price of this item and start to calculate the total cost to the purchaser

 ItemCatalogData theItemData =
    (ItemCatalogData) ItemCatalogDataHome.findByPrimaryKey(theCatalogNumber);

int totalcost = theItemData.getItemPrice();

// ...     some other processing, etc. in the interim
// ...
// ...

// Add the shipping costs
totalcost = totalcost + theItemData.getShippingCost(theDestinationPostalCode);
```

At application assembly time, the customer sets the EJB caching parameters for this bean as follows:

- ActivateAt = ONCE
- LoadAt = DAILY
- ReloadInterval = 2200

Note: The reloadInterval and reloadingEnabled attributes of the IBM deployment descriptor extensions, including both the WAR file extension (WEB-INF/ibm-web-ext.xmi) and the application extension (META-INF/ibm-application-ext.xmi) were deprecated.

On the first call to the getItemPrice() method after 22:00 each night, the EJB container reloads the pricing information from the database. If the clock strikes 22:00 between the call to getItemPrice() and getShippingCost(), the getShippingCost() method still returns the value it had before any changes to the database that might have occurred at 22:00, since the first method invocation in this transaction occurred before 22:00. Thus, the item price and shipping cost used remain in sync with each other.

What to do next

Assemble the beans in one or more EJB modules. See the topic Assembling EJB modules, or Assembling EJB 3.x modules if you are using EJB 3.x beans.

Developing message-driven beans

You can develop a bean implementation class for a message-driven bean as introduced by the Enterprise JavaBeans specification. A message-driven bean (MDB) is a message consumer that implements business logic and runs on the server.

Before you begin

Determine the messaging model you want for your application regarding use of topics, queues, producers and consumers, publish or subscribe, and so on. You can refer to the message-driven bean component contract that is described in the Enterprise JavaBeans™ specification.

About this task

A message-driven bean (MDB) is a consumer of messages from a Java Message Service (JMS) provider. An MDB is invoked on arrival of a message at the destination or endpoint that the MDB services. MDB instances are anonymous, and therefore, all instances are equivalent when not actively servicing a client message. The container controls the life cycle of bean instances, which hold no state that is visible to a client.

The following example is a basic message-driven bean:

```
@MessageDriven(activationConfig={
                @ActivationConfigProperty(propertyName="destination",     propertyValue="myDestination"),
                @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
public class MsgBean implements javax.jms.MessageListener {

  public void onMessage(javax.jms.Message msg) {

      String receivedMsg = ((TextMessage) msg).getText();
      System.out.println("Received message: " + receivedMsg);

   }

}
```

As with other enterprise bean types, you can also declare metadata for message-driven beans in the deployment descriptor rather than using annotations, for example:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar id="EJBJar_1060639024453" version="3.0"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
      metadata-complete="false">
  <enterprise-beans>

    <message-driven>

      <ejb-name>MsgBean</ejb-name>
      <ejb-class>com.acme.ejb.MsgBean</ejb-class>
      <activation-config>
         <activation-config-property>
            <activation-config-property-name>destination</activation-config-property-name>
            <activation-config-property-value>myDestination</activation-config-property-value>
         </activation-config-property>
         <activation-config-property>
           <activation-config-property-name>destinationType</activation-config-property-name>
           <activation-config-property-value>javax.jms.Queue</activation-config-property-value>
        </activation-config-property>
      </activation-config>
```

```
    </message-driven>

  </enterprise-beans>
</ejb-jar>
```

Procedure

- Code the business logic of the message-driven bean, which must implement the appropriate message listener interface defined by the messaging type; for example, javax.jms.MessageListener. The business logic is invoked when the message listener method of the MDB is called to service a message; for example, `MessageListener.onMessage()`. If the MDB implements more than one interface, denote the message listener interface by coding the `messageListenerInterface` attribute of the `MessageDriven` annotation, or by coding the `<messaging-type>` element of the `message-driven` deployment descriptor element. You do not have to specify which is the message listener interface, as long as there is only one interface other than `java.io.Serializable`, `java.io.Externalizable`, or any of the `javax.ejb` package interfaces.
- You can optionally define message destination references on any type of enterprise bean. A message destination reference is a logical name by which an enterprise bean can refer to a message destination. The `Resource` annotation is used to inject a message destination reference, for example:

  ```
  @Resource (name="jms/Outlet", type=javax.jms.Queue) Queue salesOutlet;
  ```

  Alternatively, you can use the `<message-destination-ref>` element in the deployment descriptor to specify the message destination reference; for example:

  ```
  <message-destination-ref>
     <message-destination-ref-name>jms/Outlet</message-destination-ref-name>
     <message-destination-type>javax.jms.Queue</message-destination-type>
     <injection-target>
       <injection-target-class>com.acme.ejb.MsgBean</injection-target-class>
       <injection-target-name>salesOutlet</injection-target-name>
    </injection-target>
  </message-destination-ref>
  ```

  The `message-destination-ref` element is similar to the `resource-env-ref` element, but also has subelements, `message-destination-usage` with possible values `Produces`, `Consumes` or `ProducesConsumes`, and `message-destination-link`. You can use the `message-destination-link` element to tie two or more message-destination-ref references in the deployment descriptor together, which allows the deployer to bind the destination for several enterprise beans all at once, to the same destination. The message-destination-link value must match the `message-destination-name` value in the `message-destination` element; for example:

  ```
  <ejb-jar>

    <enterprise-beans>

      <session>
        <ejb-name>OutletBean</display-name>
        ...
        <message-destination-ref>
          <message-destination-ref-name>jms/target</message-destination-ref-name>
          <message-destination-type>javax.jms.Queue</message-destination-type>
          <message-destination-usage>Produces</message-destination-usage>
          <message-destination-link>destination</message-destination-link>
        </message-destination-ref>
        ...
      </session>

      <session>
        <ejb-name>InletBean</display-name>
        ...
        <message-destination-ref>
          <message-destination-ref-name>jms/source</message-destination-ref-name>
          <message-destination-type>javax.jms.Queue</message-destination-type>
          <message-destination-usage>Consumes</message-destination-usage>
          <message-destination-link>destination</message-destination-link>
        </message-destination-ref>
        ...
      </session>
  ```

```
    <message-driven>
      <ejb-name>InletBean</display-name>
      ...

      <ejb-name>MsgBean</ejb-name>
      <ejb-class>com.acme.MsgBean</ejb-class>
      <messaging-type>javax.jms.MessageListener</messaging-type>
      <message-destination-type>javax.jms.Queue</message-destination-type>
      <message-destination-link>destination</message-destination-link>

      ...
    </message-driven>

  </enterprise-beans>
      ...
  <assembly-descriptor>
    ...
    <message-destination>
      <message-destination-name>destination</message-destination-name>
    </message-destination>
    ...
  </assembly-descriptor>

</ejb-jar>
```

The `message-destination-link` element can refer to a destination that is defined in a different Java archive (JAR) file within the same application, as with an `ejb-link` element. For example, to link to the destination, `ProduceQueue`, defined in the `grocery.jar` file, enter the following line in the deployment descriptor:

```
<message-destination-link>grocery.jar#ProduceQueue</message-destination-link>
```

- As with any enterprise bean, you can package a message-driven bean in a JAR file, or in a web application archive (WAR) file.

Results

You developed a simple message-driven bean, along with some deployment and packaging options.

What to do next

Read related information about designing an enterprise application that uses message-driven beans.

Enterprise bean development best practices

Use the following guidelines when designing and developing enterprise beans.

- Use a stateless session bean to act as the entry point for business logic.
- Entity beans should use container-managed persistence.
- In an Enterprise JavaBeans (EJB) Version 2.x and later version environments, use local interfaces to improve communication between enterprise beans in the same Java virtual machine.

  Local calls avoid the overhead of RMI/IIOP and use pass-by-reference semantics instead of pass-by-value. For each call, the caller and callee beans share the state of arguments. EJB 2.x and later beans can have both a local and remote interface, but more typically, have one or the other.
- For communicating with remote clients, provide remote and remote home interfaces. For communicating with local clients like servlets, entity beans, and message-driven beans, provide local and local home interfaces.

Batched commands for container managed persistence

From JDBC 2.0 on, **`PreparedStatement`** objects can maintain a list of commands that can be submitted together as a batch. Instead of multiple database round trips, there is only one database round trip for all the batched persistence requests.

You can enable the use of this feature for EJB container managed persistence (CMP). When you do, the run time defers **ejbStore/ejbCreate/ejbRemove** or the equivalent database persistence requests (insert/update/delete) until they are needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. When the persistence operation finally happens, run time accumulates the database requests and uses JDBC **PreparedStatement** batch operation to make a single JDBC call for multiple rows of the same operation.

The product enables you to make the same settings using assembly tools.

Deferred Create for container managed persistence

For CMP during the **ejbCreate**, the container can create the representation of the entity in the database immediately, or defer it to a later time.

You can turn this option on from the EJB CMP side. When you choose this option, the runtime defers **ejbCreate**, or the equivalent database persistence request, until it is needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. By doing this you can reduce two round trips for the newly created entity (insert and update) to one (insert).

The product enables you to make the same settings using assembly tools. Review the assembly tools information center at http://publib.boulder.ibm.com/infocenter/radhelp/v7r5mbeta/topic/com.ibm.jee5.doc/topics/cejb3.html

WebSphere extensions to the Enterprise JavaBeans specification

This topic outlines extensions to the Enterprise JavaBeans (EJB) specification provided with the product.

Inheritance in enterprise beans

In the Java language, *inheritance* is the creation of a new class from an existing class or a new interface from an existing interface. This product supports two forms of inheritance: standard class inheritance and EJB inheritance.

In standard class inheritance, the home interface, remote interface, or enterprise bean class inherits properties and methods from base classes that are not themselves enterprise bean classes or interfaces.

By contrast in enterprise bean inheritance, an enterprise bean inherits properties, such as container-managed persistence (CMP) fields and container-managed relationship (CMR) fields, methods, and method-level control descriptor attributes from another enterprise bean.

For more information, see the documentation for the assembly tools.

Optimistic concurrency control for container-managed persistence

This product supports optimistic concurrency control of data access. For more information, see the topic about concurrency control.

Access intents for EJB persistence

This product supports the application of named data-access policies.

Sequence grouping for container-managed persistence

By designating CMP sequence groups for entity beans, you can prevent certain types of database-related exceptions from occurring during the run time of your EJB application. Within each group you specify the

order in which the beans update your relational database tables. For instructions, see the topic about setting the run time for CMP sequence groups.

Performance enhancements

Through the lifetime-in-cache settings, this product provides a way for you to improve performance for beans that are only occasionally updated. For more information, see the topic about entity bean assembly settings in the assembly tool documentation.

Some enterprise beans created with the assembly tools can utilize *read-ahead* for loading a bean and its related beans in a single database operation. An entire object graph or any part of the graph can be preloaded by configuring a finder method to use read-ahead.

Assembly and deployment extensions

This product supports IBM extensions of assembly and deployment options.

Setting the run time for batched commands with JVM arguments

This article explains how to set the run time for batched commands with JVM arguments.

Procedure

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.pm.batch=true*.

Setting the run time for deferred create with JVM arguments

For Container Managed Persistence (CMP) to happen during the ejbCreate, the Enterprise JavaBeans (EJB) container can create the representation of the entity in the database immediately, or defer it to a later time.

About this task

When you choose the defer option, the run time defers ejbCreate, or the equivalent database persistence request, until it is needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. By doing this you can reduce two round trips for the newly created entity (insert and update) to one (insert).

Procedure

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.pm.deferredcreate=true*.

Setting persistence manager cache invalidation

To set persistence manager cache invalidation, follow these steps.

Procedure

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Server Infrastructure area, select **Java and Process Management**.
6. Select **Process Definition**.
7. In the Additional Properties area, select **Java Virtual Machine**.
8. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.ejbpersistence.cacheinvalidation=true*.

Setting the system property to enable remote EJB clients to receive nested or root-cause exceptions

You might want to code your application to perform a given action if a certain kind of exception is the root-cause of a failure and is nested within the exception that you receive. The default behavior in the product might mask a nested or root-cause exception in your application.

About this task

The Enterprise JavaBeans (EJB) container creates a TransactionRolledbackException exception for a remote client when it can create a RemoteException exception instead. With the RemoteException exception, the container does not lose the ability to have root-cause information nested inside the exception.

You can set the following Java virtual machine (JVM) system property to **true** through the administrative console for the product: com.ibm.websphere.ejbcontainer.includeRootExceptionOnRollback This change enables the remote client to receive nested exceptions when a rollback occurs.

Note: This property is applicable only for scenarios where the transaction in which the bean method is running was started by the container for this specific method invocation. All of the other scenarios must result in a TransactionRollBackException exception according to the EJB specification.

Procedure

1. Open the administrative console.
2. Select **Servers**.
3. Select **Servers > Application servers > server_name**.
4. Under Server infrastructure, select **Java and Process Management > Process Definition**.
5. Under Additional properties, select **Java virtual machine > Custom properties > New**.
6. In the **Name** entry field, type **com.ibm.websphere.ejbcontainer.includeRootExceptionOnRollback**.
7. In the **Value** entry field, type **true**.
8. Select **OK**.

Unknown primary-key class

When writing an entity bean, the minimum requirements usually include a primary-key class. However, in some cases you might choose not to specify the primary-key class for an entity bean with container-managed persistence (CMP).

Perhaps there is no obvious primary key, or you want to allow the deployer to select the primary key fields at deployment time. The primary key type is usually derived from the type used by the database system that stores the entity objects, and you might not know what this key is.

So, the *unknown key type* is actually a type chosen at deployment time, making it changeable each time the bean is deployed. Your client code must deal with this key as type *Object*.

Currently, WebSphere Application Server supports top-down mapping and enables the deployer to choose *String* keys generated at the application server.

Developing applications using the embeddable EJB container

Use this task to develop applications using the embeddable Enterprise JavaBeans (EJB) container. Applications running in the embeddable container start faster and require a smaller footprint than when running in the full application server. It is an optimal environment for quickly developing and testing applications that might eventually run on the application server.

Before you begin

To create an embeddable EJB container application, you must set your development environment to use Version 6.0 of Java Development Kit (JDK) Version 1.6. Your development environment must also include the WebSphere embeddable container Java archive (JAR) file in the class path. The `com.ibm.ws.ejb.embeddableContainer_8.0.0.jar` file is located in the `\runtimes` directory under the installation directory of WebSphere Application Server.

Restriction: The EJB thin client, `com.ibm.ws.ejb.thinclient_8.0.0.jar`, and the EJB embeddable JAR file, `com.ibm.ws.ejb.embeddableContainer_8.0.0.jar`, cannot coexist in the same class path.

If your beans use the javax.annotation.Resource annotation with the lookup attribute, which is new in EJB 3.1, you must also use the Java Endorsed Standards Override Mechanism to override the javax.annotation.Resource API that is available in the JDK on your system. Copy the `app_server_root\runtimes\endorsed\endorsed_apis_8.0.0.jar` file into a target directory of your choice. Use the **`java.endorsed.dirs`** property on the Java command to specify your directory that contains the copied JAR file.

Procedure

1. Create an EJB 3.X module. When creating this module, you must ensure that it only contains features that are supported by WebSphere embeddable container. For a full list of supported functions, see the “Embeddable EJB container functions” on page 372 topic. Ensure that the EJB modules are on the class path of the embeddable container. You can package the EJB module as directories of classes or as EJB JAR files.
2. Create the main class that launches the embeddable container and starts methods on the enterprise beans. Use the javax.ejb.EJBContainer class to create an instance of the embeddable container (optionally passing container configuration parameters), get the container naming context, and close the embeddable container.

   The following sample code illustrates usage of the embeddable container:

```
//EmbeddableContainerSample.java
import java.util.HashMap;
import java.util.Map;
import javax.ejb.embeddable.EJBContainer;
import my.pkg.MyBeanIface; // this is the local business interface of the
                           // enterprise bean

public class EmbeddableContainerSample {

   public static void main(String[] args) throws Throwable {
```

```
        // Create a properties map to pass to the embeddable container:
        Map<String,Object> properties = new HashMap<String,Object>();

        // Specify that you want to use the WebSphere embeddable container:
        properties.put(EJBContainer.PROVIDER,
            "com.ibm.websphere.ejbcontainer.EmbeddableContainerProvider");

        // Create the container instance, passing it the properties map:
        EJBContainer ec = EJBContainer.createEJBContainer(properties);

        // Use the container context to look up a bean:
        MyBeanIface bean = ec.getContext().lookup(
            "java:global/MyEJBModule/MyBean!my.pkg.MyBeanIface");

        // Invoke a method on the bean instance:
        bean.doStuff();

        ...


        // Close the embeddable container:
        ec.close();

    }
}
```

In this sample code, you created an instance of an embeddable container by specifying the EJBContainer.PROVIDER property to the com.ibm.websphere.ejbcontainer.EmbeddableContainerProvider class, and passing that property to the EJBContainer.createEJBContainer method. You used the container naming context to look up a local enterprise bean, for example, MyBean. The lookup uses the portable global naming syntax.

This sample code relies on the embeddable container to automatically scan the class path to find the EJB module, MyEJBModule. Alternatively, you could have specified the modules you wanted to start using the EJBContainer.MODULES property. Use this property to specify a string or string array of module names that must exist in the JVM class path.

You can also specify a file or file array of modules that do not exist in the class path. This file or file array approach might require you to modify the context class loader on the current thread, if these modules require additional libraries that are also not on the JVM class path.

The following code sample illustrates how to start the embeddable container using a file array.

```
...
// Create the properties object to pass to the embeddable container:
Map<String,Object> props = new HashMap<String,Object>();

// Specify the EJB modules to start when creating the container:
File[] ejbModules = new File[2];
ejbModules[0] = new File("/home/myusername/ejbs/ShoppingCartEJB.jar");
ejbModules[1] = new File("/home/myusername/ejbs/OnlineCatalogEJB.jar");
props.put(EJBContainer.MODULES, ejbModules);

// In this example, both of these modules rely on code in a shared library.
// In order for the embeddable container to load the shared library, the
// context classloader must be able to load that shared library.

// Set up the context classloader so that it can load the shared library:
File sharedLibUtilityFile = new File("/home/myusername/ejbs/SharedLib.jar");
ClassLoader oldCL = Thread.currentThread().getContextClassLoader();
ClassLoader newCL = new URLClassLoader(new URL[]{
   sharedLibUtilityFile.toURI().toURL()}, oldCL);
Thread.currentThread().setContextClassLoader(newCL);

// Now, create the embeddable container, passing it the properties map:
```

```
EJBContainer ec = EJBContainer.createEJBContainer(props);

// Invoke an EJB loaded by the embeddable container:
...
```

After looking up the bean instance, start methods on it. When you finish the container-related tasks, close the container, which starts the bean methods marked as PreDestroy and closes the embeddable container. Close the embeddable container instance before creating a new one.

3. Customize the embeddable container. You can use properties to customize the embeddable EJB container run time. For a full list of supported properties, see the topic Embeddable EJB container custom properties.
4. If you want your application to use resources like data sources, then you can create and configure those resources in the properties map passed to the embeddable container or in a properties file.

   One common usage of the embeddable EJB container is to test applications that eventually run in the application server. Many of these applications rely on JDBC data sources that are configured in the server using the administrative console or **wsadmin** scripting tool. Since these tools do not exist in the embeddable container, you can configure the WebSphere embeddable container to provide these resources by passing properties to it.

   The data source configuration properties can also be stored in a properties file. The embeddable container automatically loads properties stored in a file called embeddable.properties in the current working directory. You can override this file location by specifying the new file location as the value of the com.ibm.websphere.embeddable.configFileName system property.

   The data source configuration properties all start with `DataSource`, and are followed by a term that identifies which data source is being configured. For example, `DataSource.myDataSource.someProperty` applies to a different data source than one named `DataSource.anotherDS.someOtherProperty`. The list of data source properties are located in the embeddable EJB container custom properties information.

   Here is an example of how your application can use data sources:

```
...
InitialContext context = new InitialContext();
DataSource ds = (DataSource) context.lookup("env/jdbc/AcctsPayableDS");
Connection conn = ds.getConnection();
// Use the connection to access the AcctsPayableDS database
...
```

   In the server, a systems administrator has created a data source and bound it in the JNDI name space to `env/jdbc/AcctsPayableDS`. Alternatively the code might have looked up the data source in a java:comp namespace, which is mapped to `env/jdbc/AcctsPayableDS` or specified an EJB field to be injected with the data source. In all cases, a data source must be bound in the namespace. Use the following code to complete this action programmatically when creating the embeddable container instance:

```
...
// Create a properties map to store embeddable container config properties
Map<String,Object> props = new HashMap<String,Object>();

// Set the JNDI name to bind this data source:
props.put("DataSource.ds1.name", "env/jdbc/AcctsPayableDS");

// Set the data source class name ; this is a required property
// This example uses a Derby JDBC driver
props.put("DataSource.ds1.dataSourceClass",
   "org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource");

// Set the database name
props.put("DataSource.ds1.databaseName", "AcctsPayableTestDB");

// Create the embeddable container instance with our custom properties
EJBContainer ec = EJBContainer.createEJBContainer(props);

// Now invoke an EJB in the embeddable container...
...
```

The preceding code creates a simple data source to an Apache Derby database called AcctsPayableTestDB and binds it at `env/jdbc/AcctsPayableDS`. You can complete this same task declaratively by putting the following text into a file called `embeddable.properties` in the current working directory of the JVM. You can also put this text in any text file, and specify that text file in the com.ibm.websphere.embeddable.configFileName system property.

```
DataSource.ds1.name=env/jdbc/AcctsPayableDS
DataSource.ds1.dataSourceClass=org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
DataSource.ds1.databaseName=AcctsPayableTestDB
```

Use resource references when developing the EJB rather than looking up data sources directly.

5. Your application can use Java EE role-based security, both declarative and programmatic, to verify your EJB role-based security. You can complete the following actions to verify EJB role-based security:
   - Define a user to be used for authorization purposes.
   - Assign users to roles that are declared in your EJB.
   - Test the use of the EJBContext methods, isCallerInRole() and getCallerPrincipal(), in your EJB.

   Here is an example of how your application can use declarative and programmatic security:

```
import java.util.HashMap;
import java.util.Map;
import javax.ejb.EJBContainer;
import my.pkg.MyBeanIface; // this is the local business interface of the
                           // enterprise bean
public class EmbeddableContainerSample {

   public static void main(String[] args) throws Throwable {
      // Create a properties map to pass to the embeddable container:
      Map<String,Object> properties = new HashMap<String,Object>();
      // Specify that you want to use the WebSphere embeddable container:
      properties.put(EJBContainer.PROVIDER,
          "com.ibm.websphere.ejbcontainer.EmbeddableContainerProvider");

      // Specify that you want security checking enabled:
      properties.put("com.ibm.websphere.securityEnabled", "true");

      // Assign the users bob, fred, and mary to the role employee:
      props.put("role.employee", "bob, fred, mary");
      // Assign the user fred to the role manager:
      props.put("role.manager", "fred");
      // The user fred will be used for the runAs role manager:
      props.put("role.runAs.manager", "fred");
      // The user fred will be used for role authorization when invoking
      // methods on the EJB:
      props.put("user.invocation", "fred");
      // Create the container instance, passing it the properties map:
      EJBContainer ec = EJBContainer.createEJBContainer(properties);
      // Use the container context to look up a bean:
      MyBeanIface bean = ec.getContext().lookup(
          "java:global/MyEJBModule/MyBean!my.pkg.MyBeanIface");
      // Invoke a method on the bean instance:
      bean.doStuff();
      ...
      // Close the embeddable container:
      ec.close();
   }
}
```

The preceding code enables security and then creates two roles, `employee` and `manager`, and three users, `bob`, `mary`, and `fred`. It then specifies that the `fred` user is used when running as the `manager` role. Before creating the embeddable container, it sets the invocation user as `fred`. This means that when EJB methods are started, they are started by `fred`, which means that if those methods require the role of `employee` or `manager`, `fred` is able to access those methods.

6. Your application can specify Local Transaction Containment (LTC) behavior per bean. For an explanation of LTC, read about local transaction containment.

Here is an example of how your application can specify the LTC resolver and unresolved action:

```
import java.util.HashMap;
import java.util.Map;
import javax.ejb.EJBContainer;
import my.pkg.MyBeanIface; // this is the local business interface
                           // of the enterprise bean

 public class EmbeddableContainerSample {

   public static void main(String[] args) throws Throwable {

      // Create a properties map to pass to the embeddable container:
      Map<String,Object> properties = new HashMap<String,Object>();
      // Specify that you want to use the WebSphere embeddable container:
      properties.put(EJBContainer.PROVIDER,
          "com.ibm.websphere.ejbcontainer.EmbeddableContainerProvider");

      // Specify that you want the LTC resolver container-at-boundary:
      properties.put("Bean.myApp1#moduleA#bean101.LocalTransaction.Resolver",
                                                  "ContainerAtBoundary");

      // Specify that you want the LTC unresolved action commit:
      properties.put("Bean.myApp1#moduleA#bean101.LocalTransaction.UnresolvedAction",
                                                  "Commit");

      // Create the container instance, passing it the properties map:
      EJBContainer ec = EJBContainer.createEJBContainer(properties);
      // Use the container context to look up a bean:
      MyBeanIface bean = ec.getContext().lookup(
          "java:global/MyEJBModule/MyBean!my.pkg.MyBeanIface");
      // Invoke a method on the bean instance:
      bean.doStuff();
      ...
      // Close the embeddable container:
      ec.close();
   }
}
```

The preceding code sets the resolver action for the specified bean to its non-default value of container-at-boundary, and causes the unresolved action to be the non-default action that commits the transaction.

If an application name is not specified when launching the embeddable container, the *<application_name>* must be omitted. However, you must use the # delimiter. For example:

```
properties.put("Bean.#moduleA#bean101.LocalTransaction.UnresolvedAction", "Commit");
```

Embeddable EJB container

The embeddable Enterprise JavaBeans (EJB) container is a container for enterprise beans that do not require a Java Platform, Enterprise Edition (Java EE).

TheWebSphere Application Server embeddable EJB container is a container for enterprise beans that does not require a Java EE server to run. The EJB programming model and the EJB container services are now available for Java Platform, Standard Edition (Java SE) server.

The following are embeddable container usage scenarios:

- EJB unit testing: developers can test their enterprise beans without needing a full server installation of WebSphere Application Server in their development environment.
- Embedding enterprise beans in Java SE applications: developers can use enterprise beans and the functionality that is provided with an EJB container, for example, dependency injection, transactions, and security in stand-alone desktop applications.

The following advantages exist when using the WebSphere embeddable EJB container:

- No server installation is necessary for EJB development, unit testing, and Java SE-based application deployment.
- The embeddable container is a much smaller footprint, in terms of disk space and main memory, than the server-based container.
- The embeddable container starts faster than the server-based container because it initializes only EJB-related components.

Be aware of the following limitations when using the embeddable container:
- Inbound RMI/IIOP calls are not supported, which means that all EJB clients must exist within the same Java virtual machine (JVM) as the embeddable container.
- Message driven beans (MDB) are not supported.
- The embeddable container cannot be clustered for high availability per workload management. For a complete list of supported functions in the WebSphere embeddable container, see the topic, Embeddable EJB container functions.

Running an embeddable container

Use this task to run an embeddable container. Applications running in the embeddable container start faster and require a smaller footprint than when running in the full application server. It is an ideal environment for quickly developing and testing applications that might eventually run in the application server.

Before you begin

Before running an embeddable container, you must have the following items ready:
- A copy of the *<app_server_root>*\runtimes\com.ibm.ws.ejb.embeddableContainer_8.5.0.jar

  Restriction: The Enterprise JavaBeans (EJB) thin client, com.ibm.ws.ejb.thinclient_8.5.0.jar, and the EJB embeddable Java archive (JAR) file, com.ibm.ws.ejb.embeddableContainer_8.5.0.jar, cannot coexist in the same class path.
- A copy of the *<app_server_root>*\runtimes\endorsed\endorsed_apis_8.5.0.jar file, if you are using the @Resource annotation with the lookup attribute
- One or more EJB modules in JAR files or class directories
- A main class that creates the embeddable container
- A Java SE Development Kit (JDK) or a Java Runtime Environment (JRE) Version 6.0 or later

About this task

The key to running the embeddable container is the class path. The class path must include all the artifacts previously listed. For example, if the main class is my.pkg.MyMainClass, and it uses enterprise beans that are stored in the MyEJBModule.jar file, the following line might run the main class that launches the embeddable container. This example assumes that all JAR files and class directories are in the current working directory.
- Windows:

```
C:\test> java -cp .;com.ibm.ws.ejb.embeddableContainer_8.5.0.jar my.pkg.MyMainClass
```
- UNIX:

```
[test]$ java -cp .:com.ibm.ws.ejb.embeddableContainer_8.5.0.jar:MyEJBModule.jar my.pkg.MyMainClass
```

It is possible to run the embeddable container without specifying all the modules on the class path. This requires the code to specify a File or File array for the MODULES property, and the context class loader for the thread creating the container instance must be able to load the specified files.

If you want to specify embeddable container properties in a text file other than embeddable.properties in the current working directory, then you must specify the com.ibm.websphere.embeddable.configFileName system property; for example:

- Windows:

```
C:\test> java -Dcom.ibm.websphere.embeddable.configFileName="C:\test\my-config.properties"
 -cp .;com.ibm.ws.ejb.embeddableContainer_8.5.0.jar;MyEJBModule.jar my.pkg.MyMainClass
```

- UNIX:

```
[test]$ java -Dcom.ibm.websphere.embeddable.configFileName="/home/myusername/test/my-config.properties"
 -cp .:com.ibm.ws.ejb.embeddableContainer_8.5.0.jar:MyEJBModule.jar my.pkg.MyMainClass
```

When developing an application using JPA in the embeddable EJB container, the class path must include the JPA thin client, `com.ibm.ws.jpa.thinclient_`*n*`.0.jar`, where *n* is the WebSphere Application Server release; for example, 8.5 for Version 8.5. The JPA thin client is located in `\runtimes` where the root directory of the installation image is located.

- Windows:

```
C:\test> java -cp .;com.ibm.ws.ejb.embeddableContainer_8.5.0.jar;%WAS_HOME%\runtimes
  \com.ibm.ws.jpa.thinclient_8.5.0.jar;MyEJBModule.jar my.pkg.MyMainClass
```

- UNIX:

```
[test]$ java -cp .:com.ibm.ws.ejb.embeddableContainer_8.5.0.jar:${WAS_HOME}/runtimes
  /com.ibm.ws.jpa.thinclient_8.5.0.jar:MyEJBModule.jar my.pkg.MyMainClass
```

Note: You can specify the Java agent mechanism to complete the dynamic enhancement at run time. For example, type the following line of code at the command prompt:

- Windows:

```
C:\test> java -javaagent:%WAS_HOME%\runtimes\com.ibm.ws.jpa.thinclient_8.5.0.jar
 -cp .;com.ibm.ws.ejb.embeddableContainer_8.5.0.jar;MyEJBModule.jar my.pkg.MyMainClass
```

- UNIX:

```
[test]$ java -javaagent:${WAS_HOME}/runtimes/com.ibm.ws.jpa.thinclient_8.5.0.jar
 -cp .:com.ibm.ws.ejb.embeddableContainer_8.5.0.jar:MyEJBModule.jar my.pkg.MyMainClass
```

To enable tracing in the embeddable container, you can specify the com.ibm.ejs.ras.lite.traceSpecification system property to a trace specification value as you would specify for the server. By default, the trace is printed to standard output, but you can redirect the output by specifying the com.ibm.ejs.ras.lite.traceFileName system property. The following example shows how you can use both system properties:

- Windows:

```
C:\test> java -Dcom.ibm.ejs.ras.lite.traceSpecification=EJBContainer=all:MetaData=all
 -Dcom.ibm.ejs.ras.lite.traceFileName=trace.log
 -cp .;com.ibm.ws.ejb.embeddableContainer_8.5.0.jar;MyEJBModule.jar my.pkg.MyMainClass
```

- UNIX:

```
[test]$ java -Dcom.ibm.ejs.ras.lite.traceSpecification=EJBContainer=all:MetaData=all
 -Dcom.ibm.ejs.ras.lite.traceFileName=trace.log
 -cp .:com.ibm.ws.ejb.embeddableContainer_8.5.0.jar:MyEJBModule.jar my.pkg.MyMainClass
```

If your beans use the javax.annotation.Resource annotation with the lookup attribute, you must also use the Java Endorsed Standards Override Mechanism to override the javax.annotation.Resource API that is available in the JDK on your system. Copy the *app_server_root*`\runtimes\endorsed\endorsed_apis_8.5.0.jar` file into a target directory of your choice. Use the **java.endorsed.dirs** property on the Java command to specify your directory that contains the copied JAR file. The following example shows how you can specify the java.endorsed.dirs property:

- Windows:

```
C:\test> java -Djava.endorsed.dirs="myTargetDirectory"
 -cp .;com.ibm.ws.ejb.embeddableContainer_8.5.0.jar;MyEJBModule.jar my.pkg.MyMainClass
```

- UNIX:

```
[test]$ java -Djava.endorsed.dirs="myTargetDirectory"
 -cp .:com.ibm.ws.ejb.embeddableContainer_8.5.0.jar:MyEJBModule.jar my.pkg.MyMainClass
```

Embeddable EJB container functions

According to the Enterprise JavaBeans (EJB) 3.1 specification, all embeddable EJB containers that vendors use must at least implement the EJB Lite subset of EJB functionality. The application server also contains additional features that support the EJB Lite subset. Refer to the EJB 3.1 specification for more information.

Attention: Container-managed authentication is only supported with the default container-managed authentication alias. For data sources, the user ID and password fields of the Java EE data source resource, or the embeddable properties data source, are used as the default container-managed authentication alias.

EJB Lite includes:

- Local (and no-interface) session beans with synchronous methods only, which include stateless, stateful, and singleton bean types.
- Declarative and programmatic security.
- Interceptors.
- Support for annotations or XML deployment descriptors, the `ejb-jar.xml` file.
- Java Persistence Architecture (JPA) 2.0.

The WebSphere embeddable container provides the following additional functions:

- Java Database Connectivity (JDBC) data source configuration, usage, and dependency injection.
- Bean validation

  To use bean validation with the embeddable EJB container, the javax.validation classes must exist in the class path. That can be done in one of two ways:
  - Include the JPA thin client that is located in the directory `${WAS_INSTALL_ROOT}\runtimes\com.ibm.ws.jpa.thinclient_8.0.0.jar` in the class path. See the topic, Running an embeddable container, and the information about JPA, for more information.
  - Include a third party bean validation provider Java archive (JAR) file in the class path of the embeddable EJB container run time.

Embeddable EJB container configuration properties

Use the following configuration properties for the embeddable Enterprise JavaBeans (EJB) container.

Table 52. Embeddable EJB container configuration properties. Use the properties for embeddable EJB container configurations.

Property	Type	Default value	Description
com.ibm.websphere.ejbcontainer.cacheSize	java.lang.Long	2 053	Number of buckets for the EJB cache.
com.ibm.websphere.ejbcontainer.cacheSweepInterval	java.lang.Long	3 000	Time between sweeps of the EJB cache to determine whether to remove entries.
com.ibm.websphere.ejbcontainer.inactivePoolCleanupInterval	java.lang.Long	30 000	Time in milliseconds for the clean up thread to wait before cleaning the inactive pool.
com.ibm.websphere.ejbcontainer.passivationDir	java.lang.String	<TempDir>	Directory to passivate stateful beans in. The user must have read and write access to the specified directory.

Table 52. Embeddable EJB container configuration properties (continued). Use the properties for embeddable EJB container configurations.

Property	Type	Default value	Description
com.ibm.websphere.embeddable.configFileName	java.lang.String	<CurrentWorkingDirectory> /embeddable.properties	File name of a properties file containing embeddable EJB container properties. When this file is processed, each property is passed to the newly created EJB container as if they were passed in programmatically. Any properties in the configuration file are overridden by properties passed in programmatically. **Attention:** It is also possible to specify this property as a system property on the command line.

The following table contains configuration properties for data sources. Each property is specific to an individual data source which allows you to configure multiple data sources with different settings. Replace *<data_source_id>* with a unique term that identifies the data source to be configured. Some properties are listed as required for each data source.

Table 53. Embeddable EJB container configuration properties for Java Database Connectivity (JDBC) data sources. Use the embeddable EJB container configuration properties for JDBC data sources.

Property	Type	Description
DataSource.*<data_source_id>*.name	java.lang.String	Required. The Java Naming and Directory Interface (JNDI) string that the container uses to bind this data source in the global namespace of the embeddable container. This string must match the JNDI lookup string used in the application.
DataSource*<data_source_id>*.className	java.lang.String	Required. The Java class name of the data source class. For testing, Apache Derby can be used if *app_server_root*/derby/lib/derby.jar is on the Java Virtual Machine classpath. The supported data source classes for Apache Derby are org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource and org.apache.derby.jdbc.EmbeddedXADataSource.
DataSource.*<data_source_id>*.connectionSharing	java.lang.String	The policy for sharing connections. Valid values are `MatchOriginalRequest` (default), `MatchCurrentState` or `None`. `MatchOriginalRequest` means that a connection request might share if it matches the originally requested settings of an existing connection. `MatchCurrentState` means that a connection request might share if it matches the current settings of an existing connection. `None` means that multiple connection requests do not share the same connection.
DataSource.*<data_source_id>*.databaseName	java.lang.String	The name of the database that this data source connects to.
DataSource.*<data_source_id>*.isolationLevel	java.lang.Integer	The transaction isolation level for connections from this data source. If unspecified, the default is provided by the JDBC driver or database. Valid values are `1` for Read Uncommitted, `2` for Read Committed, `4` for Repeatable Read, or `8` for Serializable. These values come from constants in java.sql.Connection and are subject to change.
DataSource.*<data_source_id>*.maxIdleTime	java.lang.Integer	The number of seconds after which the connection pool can close an unused connection.
DataSource.*<data_source_id>*.maxPoolSize	java.lang.Integer	The maximum number of connections that are created for this data source. After this number of simultaneous connections are in use, future requests to get a connection from this data source are blocked until one or more of the in-use connections have been returned to the pool.
DataSource.*<data_source_id>*.maxStatements	java.lang.Integer	The maximum number of statements cached by the connection pool. This value is divided by the *maxPoolSize* value to determine the number of statements that can be cached for each connection in the pool. A value of `0` disables statement caching.
DataSource.*<data_source_id>*.minPoolSize	java.lang.Integer	The minimum number of connections to keep in the connection pool for this data source. If no connections are in use, the connection pool might discard connections until the pool size reaches this setting. This setting must be non-negative.
DataSource.*<data_source_id>*.password	java.lang.String	Password for the given `user` when accessing the database. Like the preceding property, this property can be omitted if the database does not have security enabled or if the user name and password are provided programmatically when creating the connection.
DataSource.*<data_source_id>*.transactional	java.lang.Boolean	Whether this data source must be enlisted in Java Transaction API (JTA) transactions. Valid values are `true` (default) or `false`.
DataSource.*<data_source_id>*.user	java.lang.String	User name for accessing the database. This property can be omitted if the database does not have security enabled or if the user name and password are provided programmatically when creating the connection.
DataSource.*<data_source_id>.<vendor_property_name _or_connection_pool_property_name>*	java.lang.String	You can also configure other properties: • Vendor-specific data source properties, such as `serverName` and `portNumber` • WebSphere Application Server data source properties, such as `userDefinedErrorMap` and `validateNewConnection` • WebSphere Application Server connection pooling properties, such as `connectionTimeout` and `purgePolicy`
DataSource.*<data_source_id>*.xaRecoveryPassword	java.lang.String	Applies to XA data sources only. Password for XA recovery.

Table 53. Embeddable EJB container configuration properties for Java Database Connectivity (JDBC) data sources (continued). Use the embeddable EJB container configuration properties for JDBC data sources.

Property	Type	Description
DataSource.<*data_source_id*>.xaRecoveryUser	java.lang.String	Applies to XA data sources only. Some databases require a user with special privileges for XA recovery. Use this property to specify a user name for XA recovery instead of the default user.

Use the following properties to configure two data sources.

```
DataSource.ds1.name=env/jdbc/ds1
DataSource.ds1.className=org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource
DataSource.ds1.transactional=true
DataSource.ds1.createDatabase=create
DataSource.ds1.databaseName=jtest1
DataSource.ds1.user=dbuser1
DataSource.ds1.password=dbpwd1
DataSource.ds1.maxPoolSize=5

DataSource.ds2.name=env/jdbc/ds2
DataSource.ds2.className=org.apache.derby.jdbc.EmbeddedXADataSource
DataSource.ds2.connectionSharing=MatchOriginalRequest
DataSource.ds2.createDatabase=create
DataSource.ds2.databaseName=jtest2
DataSource.ds2.user=dbuser2
DataSource.ds2.password=dbpwd2
DataSource.ds2.maxPoolSize=10
DataSource.ds2.minPoolSize=1

DataSource.ds3.name=env/jdbc/ds3
DataSource.ds3.className=com.ibm.db2.jcc.DB2XADataSource
DataSource.ds3.driverType=4
DataSource.ds3.databaseName=DB2COPY1
DataSource.ds3.serverName=mydb2.test.ibm.com
DataSource.ds3.portName=50000
DataSource.ds3.user=dbuser1
DataSource.ds3.password=dbpwd1
```

The first data source, ds1, is bound in the namespace under the name, env/jdbc/ds1, and provides connection pooling for up to five connections to an Apache Derby database. The second data source, ds2, is bound in the namespace at env/jdbc/ds2 and provides XA-compliant connection pooling to an Apache Derby database.

The following table contains configuration properties for EJB bindings. For each property, replace:

- *<app>* with the application name.

  The application name is, by default, the empty string. However the application name can be specified using the EJBContainer.APP_NAME property when calling createEJBContainer.
- *<module>* with the module name.

  The module name is either specified in ejb-jar.xml as <module-name>, is the name of the JAR file without the ".jar" suffix, or is the name of the directory containing the module.
- *<ejb>* with the name of the EJB.

  The name of the EJB is specified in ejb-jar.xml as <ejb-name>, or using the name element when using annotations; for example, @Stateless(name="TestBean"). If no name is specified in the EJB annotation, the simple class name is used; for example, TestBean for the class com.ibm.test.TestBean.
- *<interceptor>* with the name of an interceptor class.
- *<ref>* with the name of the resource reference, EJB reference, or environment entry.

  The reference can be specified in ejb-jar.xml; for example, <res-ref-name>jdbc/mydsref</res-ref-name>, or <ejb-ref-name>java:module/env/myejbref</ejb-ref-name>. Alternatively, the reference can be specified using an annotation; for example @Resource(name="jdbc/mydsref") or @EJB(name="java:module/env/myenvref").

If an application name is not specified when launching the embeddable container, the *<app>* must be omitted. However, you must use the # delimiter. For example, `Bean.#<module>#<bean>.ResourceRef.BindingName.`

Table 54. Embeddable EJB container configuration properties for reference bindings. Use the embeddable EJB container configuration properties for reference bindings.

Property	Type	Description
Bean.*<app>*#*<module>*#*<ejb>*.ResourceRef.BindingName.*<ref>* or Interceptor.*<app>*#*<module>*#*<interceptor>*.ResourceRef.BindingName.*<ref>*	java.lang.String	The JNDI string to use when the resource reference is looked up or injected. This must be the JNDI name of a configured data source.
Bean.*<app>*#*<module>*#*<ejb>*.EJBRef.BindingName.*<ref-name>* or Interceptor.*<app>*#*<module>*#*<interceptor>*.EJBRef.BindingName.*<ref>*	java.lang.String	The JNDI string to use when the EJB reference is looked up or injected. This must be the java:global, java:app, or java:module JNDI string of an EJB in the embeddable EJB container.
Bean.*<app>*#*<module>*#*<ejb>*.EnvEntry.Value.*<ref-name>* or Interceptor.*<app>*#*<module>*#*<interceptor>*.EnvEntry.Value.*<ref>*	java.lang.String	The value to use when the environment entry is looked up or injected. This property overrides the value specified by env-entry-value. The value must be valid for the type of the environment entry.
Bean.*<app>*#*<module>*#*<ejb>*.EnvEntry.BindingName.*<ref>* or Interceptor.*<app>*#*<module>*#*<interceptor>*.EnvEntry.BindingName.*<ref>*	java.lang.String	The JNDI string to use when the environment entry is looked up or injected. This must be the java:global, java:app, or java:module JNDI string of another environment entry in the same embeddable EJB container.
Bean.*<app>*#*<module>*#*<ejb>*.DataSource.BindingName.*<ref>* or Interceptor.*<app>*#*<module>*#*<interceptor>*.DataSource.BindingName.*<ref>*	java.lang.String	The JNDI string to use when the data source is looked up or injected. This must either be the JNDI name of a configured data source or the java:global, java:app, or java:module JNDI string of another data source in the same embeddable EJB container. Application developers should use resource references rather than looking up a data source directly. If an application is coded to look up data sources directly, you can use this property to override data source definitions included in the application. See the information center for more information about data source definitions.

Use the following properties to configure the bindings for an EJB with two references and an interceptor with an environment entry:

```
Bean.#TestModule#TestBean.ResourceRef.BindingName.jdbc/dsref=env/jdbc/ds1
Bean.#TestModule#TestBean.EJBRef.BindingName.ejb/Cart=java:global/CartModule/CartBean
Interceptor.#TestModule#com.ibm.example.LoggerInterceptor.EnvEntry.Value.logFile=/tmp/output.log
```

Note: Use the following properties to override java:module/env/TestDataSource that is defined in an EJB named TestBean in an EJB module named TestModule, with jdbc/MyDataSource that is defined in embeddable properties:

```
Bean.#TestModule#TestBean.DataSource.BindingName.java\:module/env/TestDataSource=jdbc/MyDataSource
```

Table 55. Embeddable EJB container configuration properties for security. Use the embeddable EJB container configuration properties for security.

Property	Type	Default value	Description
com.ibm.websphere.securityEnabled	java.lang.String	false	Determines whether security roles are checked. Valid values are false (default) or true. If true, then security roles are checked; if false, then security roles are not checked.
role.<role_name>	java.lang.String		Maps an EJB role to one or more users. The <role_name> is a role assigned to a method, either through the annotation @RolesAllowed or through the deployment descriptor <method-permission>. The value is a string of comma-delimited user names, for example “bob, mary, john”. The users in the list are allowed to run methods that require the <role_name>.
role.runAs.<role_name>	java.lang.String		Maps one EJB role to one user. The <role_name> is a role assigned to a bean or method, either through the annotation @RunAs or through the deployment descriptor <run-as>. The value is a single user name. The user name is used for any authorization that is required by the bean while it is running.

Table 55. Embeddable EJB container configuration properties for security (continued). Use the embeddable EJB container configuration properties for security.

Property	Type	Default value	Description
user.invocation	java.lang.String		Defines the user that might be used for authorization when the bean is invoked. The value is a single user name. The container checks that this user is mapped to a role that is allowed to run any executed method.

The following table contains configuration properties for the Local Transaction Containment (LTC) behavior. For an explanation of LTC, read about local transaction containment. Each property is specific to a bean.

If an application name is not specified when launching the embeddable container, the *<application_name>* must be omitted. However, you must use the # delimiter. For example, `Bean.#<module_name>#<bean_name>LocalTransaction.Resolver`.

Table 56. Embeddable EJB container configuration properties for Local Transaction Containment. Use the embeddable EJB container configuration properties for Local Transaction Containment.

Property	Type	Default value	Description
Bean.*<application_name>#<module_name>#<bean_name>*.LocalTransaction.Resolver	java.lang.String	Application	Determines the entity that is responsible for local transaction resolution. Valid values are Application (default) or ContainerAtBoundary.
Bean.*<application_name>#<module_name>#<bean_name>*.LocalTransaction.UnresolvedAction	java.lang.String	Rollback	Determines the action taken for unresolved local transactions. Valid values are Rollback (default) or Commit.

The following table contains configuration properties for XA behavior.

Table 57. Embeddable EJB configuration properties for XA. Use the embeddable EJB configuration properties for XA.

Property	Type	Default value	Description
com.ibm.websphere.tx.acceptHeuristicHazard	java.lang.String	false	Specifies whether last participant support is enabled for all modules. The default value is `false`.
com.ibm.websphere.tx.auditRecovery	java.lang.String	true	Specifies whether recovery processing outputs audit messages, which indicate XA resource and XID processing during recovery. When no audit recovery is specified, only a single recovery message is output along with the number of transactions that are recovered, unless an error occurs.
com.ibm.websphere.tx.clientInactivityTimeout	java.lang.String	0	Specifies the maximum duration, in seconds, between transactional requests. Any period of client inactivity that exceeds this timeout value results in the transaction being rolled back. The default setting, `0`, means that no limit exists.
com.ibm.websphere.tx.enableLoggingForHeuristicReporting	java.lang.String	false	This property enables logging for heuristic reporting. If last participant support is enabled, reporting of heuristic outcomes that might occur when the server becomes unavailable requires additional information to be written to the transaction log. If enabled, one additional log write is completed for any transaction that involves both one-phase and two-phase commit resources. No additional records are written for transactions that do not involve a one-phase commit resource.
com.ibm.websphere.tx.heuristicRetryLimit	java.lang.String	0	Specifies the number of times that the transaction service retries a completion signal such as commit or rollback. Retries occur after a transient exception from a resource manager. The default value, `0`, indicates no limit to the number of retries.
com.ibm.websphere.tx.heuristicRetryWait	java.lang.String	0	Specifies the number of seconds that the transaction service waits before retrying a completion signal, such as commit or rollback, after a transient exception from a resource manager.
com.ibm.websphere.tx.LPSHeuristicCompletion	java.lang.String	ROLLBACK (case insensitive)	The heuristic completion action to be taken by the transaction service in a transaction with last participant support when the outcome of the one-phase commit resource is unknown. The values ROLLBACK or COMMIT cause the two-phase commit resources to be completed accordingly. The setting, MANUAL, means that the transaction service takes no action and leave the two-phase commit resources in-doubt. The default value is ROLLBACK.

Table 57. Embeddable EJB configuration properties for XA (continued). Use the embeddable EJB configuration properties for XA.

Property	Type	Default value	Description
com.ibm.websphere.tx.maximumTransactionTimeout	java.lang.String	300	Specifies, in seconds, the upper limit of the transaction timeout value. This timeout value constrains the upper limit of all other transaction timeout values.
com.ibm.websphere.tx.totalTranLifetimeTimeout	java.lang.String	120	Specifies the default maximum time, in seconds, allowed for a transaction before the transaction service initiates timeout. Any transaction that does not begin completion processing before this timeout occurs is rolled back.
com.ibm.websphere.tx.tranLogDirectory	java.lang.String	profiles*server name*\tranlog	Specifies the name of a directory for this server where the transaction service stores log files for recovery.
com.ibm.websphere.tx.tranLogSize	java.lang.String	1024	Specifies the size, in kilobytes, of transaction log files. The minimum file size is 64KB. The default value sets the file size to 1MB.

Configuring EJB 3.1 session bean methods to be asynchronous

Use this task to configure Enterprise JavaBeans (EJB) 3.1 session bean methods to run asynchronously. You can make some or all of your bean methods asynchronous.

Before you begin

Attention: In EJB 3.1 modules, you can set one or more session bean methods to be asynchronous, broadening parallel processing in your application.

- If you are not already familiar with EJB 3.1 asynchronous methods, read about EJB 3.1 asynchronous methods, client programming model for EJB asynchronous methods, bean implementation programming model for EJB asynchronous methods, and EJB container work manager for asynchronous methods. The topics provide an overview of EJB 3.1 asynchronous methods, describe the client and bean implementation programming models, and discuss the work manager that the EJB container uses to dispatch asynchronous methods.
- Develop a new EJB 3.1 session bean for your application, or change an existing session bean so that it conforms to the EJB 3.1 programming model requirements for asynchronous methods. For general information, see information about developing enterprise beans.

About this task

After you have developed a session bean, complete the following steps to make one or more of the bean methods asynchronous.

Procedure

1. Specify one or more methods of the bean implementation class as asynchronous. This can be accomplished by adding @Asynchronous annotations in your bean source code, by adding <async-method> stanzas in your module deployment descriptor, or by adding a combination of both annotations and deployment descriptor stanzas. You can apply the @Asynchronous annotation or its superclasses only, to your bean implementation class. It cannot be applied to interface classes. Also, when the annotation is applied at the class level, all methods of that class are asynchronous. Likewise, all methods of a bean can be configured as asynchronous by applying "*" as the <method-name> in your deployment descriptor.

   See the following examples of applying the @Asynchronous annotation:

   - Apply the @Asynchronous annotation to one method of a bean with a no-interface view. In this example, the m1 method is synchronous and the m2 method is asynchronous.

```
@Stateless @LocalBean
public class MyLocalBean {

   public void m1() {

      // method code
```

```
   }

   @Asynchronous
   public Future<String> m2() {

      // method code

      return new javax.ejb.AsyncResult("Hello, Async World!");
   }
}
```

Important: The javax.ejb.AsyncResult<V> object is a convenience implementation of the Future<V> interface. See the API documentation for more details.

- Apply the @Asynchronous annotation to the class level of a bean class. In this example, both the m1 method and the m2 method are asynchronous on this no-interface view bean.

```
@Stateless @LocalBean @Asynchronouspublic class MyLocalBean {

   public void m1() {

      // method code
   }

   public Future<String> m2() {

      // method code

      return new javax.ejb.AsyncResult("Hello, Async World!");
   }

}
```

- Apply the @Asynchronous annotation to one method of a bean implementation class. In this example, the m1 method is synchronous and the m2 method is asynchronous. This example also demonstrates how the return types might differ between the business interface and the implementation class.

```
public interface MyIntf {

   public void m1();

   public Future<Integer> m2();

}
@Stateless @Local(MyIntf.class)
public class MyBean {

   public void m1() {

      // method code
   }

   @Asynchronous
   public Integer m2() {

      // method code

      return new Integer(3);
   }
}
```

- Apply the @Asynchronous annotation to the class level of a bean implementation class. In this example, both the m1 method and the m2 method are asynchronous.

```
@Stateless @Local(MyIntf.class) @Asynchronous
public class MyBean {
```

```
   public void m1() {

      // method code
   }

   public Integer m2() {

      // method code

      return new Integer(8);
   }
}
```

See the following examples of modifying the EJB module deployment descriptor, `ejb-jar.xml`:

- In this example all business methods of the FullAyncBean bean implementation class and its superclasses are configured as asynchronous with the wildcard (*) method-name element.

```
<session>
   <display-name>FullAsyncEJB</display-name>
   <ejb-name>FullAsyncBean</ejb-name>
   <business-local>com.ibm.sample.async.ejb.FullAsyncIntf</business-local>
   <ejb-class>com.ibm.sample.async.ejb.FullAsyncBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>*</method-name>
   </async-method>
</session>
```

- In this example only the specified methods and signatures -- all methods named m1 and the method m2 with a single String parameter -- are configured as asynchronous on the PartiallyAsyncBean bean implementation class.

```
<session>
   <display-name>PartiallyAsyncEJB</display-name>
   <ejb-name>PartiallyAsyncEJB</ejb-name>
   <business-local>com.ibm.sample.async.ejb.PartiallyAsyncIntf</business-local>
   <ejb-class>com.ibm.sample.async.ejb.PartiallyAsyncBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>m1</method-name>
   </async-method>
   <async-method>
      <method-name>m2</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </async-method>
</session>
```

See the following examples of applying a combination of the @Asynchronous annotation in the bean source code and modifying the EJB module deployment descriptor, `ejb-jar.xml`:

- In this example the @Asynchronous annotation configures method m2 to be asynchronous, and the deployment descriptor configures method m1 to also be an asynchronous method.

```
@Stateless @LocalBean
public class MyLocalBean {

   public void m1() {

      // method code
   }

   @Asynchronous
   public Future<String> m2() {

      // method code

      return new javax.ejb.AsyncResult("Hello, Async World!");
   }
}
```

```
<session>
   <display-name>MyLocalEJB</display-name>
   <ejb-name>MyLocalEJB</ejb-name>
   <local-bean/>
   <ejb-class>com.ibm.sample.async.ejb.MyLocalBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>m1</method-name>
   </async-method>
</session>
```

- In this example the @Asynchronous annotation for method m2 is ignored because the deployment descriptor header contains the metadata-complete="true" flag. This flag causes configuration information to only be taken from the deployment descriptor elements. The result is that only method m1 of the MyLocalBean implementation is configured to be asynchronous.

```
@Stateless @LocalBean
public class MyLocalBean {

   public void m1() {

      // method code
   }

   @Asynchronous
   public Future<String> m2() {

      // method code

      return new javax.ejb.AsyncResult("Hello, Async World!");
   }
}


<ejb-jar id="ejb-jar_ID" ...
metadata-complete="true" version="3.1">
...
<session>
   <display-name>MyLocalEJB</display-name>
   <ejb-name>MyLocalEJB</ejb-name>
   <local-bean/>
   <ejb-class>com.ibm.sample.async.ejb.MyLocalBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>m1</method-name>
   </async-method>
</session>
...
</ejb-jar>
```

2. Verify that the transaction attribute applied to any asynchronous method is either REQUIRED, REQUIRES_NEW, or NOT_SUPPORTED. These transaction attribute types are the only transaction attribute types supported on asynchronous methods. You can complete this action by either applying @TransactionAttribute annotations in the bean source code, by adding <container-transaction> stanzas in the `ejb-jar.xml` file, or by adding a combination of both annotations and <container-transaction> stanzas in the deployment descriptor.

   See the following example of setting the transaction attribute of an asynchronous method using annotations:

```
@Singleton @LocalBean
public class FullAsyncBean {
 @Asynchronous
 @TransactionAttribute(REQUIRED) // the default; specified for illustration
 public void m1() {
  // ...
 }

 @Asynchronous
```

```
 @TransactionAttribute(NOT_SUPPORTED)
 public void m2() {
  // ...
 }

 @Asynchronous
 @TransactionAttribute(REQUIRES_NEW)
 public void m3() {
  // ...
 }

 // ...
}
```

See the following example of setting the transaction attribute of an asynchronous method using the XML deployment descriptor:

```
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>FullAsyncBean</ejb-name>
        <method-name>m1</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>FullAsyncBean</ejb-name>
        <method-name>m2</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>FullAsyncBean</ejb-name>
        <method-name>m3</method-name>
      </method>
      <trans-attribute>RequiresNew</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
```

See the following example of using a combination of both annotations and the XML deployment descriptor to configure the transaction attributes of a bean. In this example the deployment descriptor stanzas for method m3 override the class level annotation. The result is that method m3 is configured as REQUIRES_NEW, while methods m1 and m2 are configured as REQUIRED:

```
@Singleton @LocalBean
@Asynchronous
@TransactionAttribute(REQUIRED) // the default; specified for illustration
public class FullAsyncBean {

 public void m1() {
  // ...
 }

 public void m2() {
  // ...
 }

 public void m3() {
  // ...
 }

 // ...
}
```

```
<assembly-descriptor>

   <container-transaction>
      <method>
         <ejb-name>FullAsyncBean</ejb-name>
         <method-name>m3</method-name>
      </method>
      <trans-attribute>RequiresNew</trans-attribute>
   </container-transaction>

</assembly-descriptor>
```

What to do next

Continue to develop additional components for your application, or if you have finished all components required by your application, assemble and deploy your application. See information about assembling EJB modules and deploying EJB modules.

When you run your application, if it fails when it first attempts to use a session bean that has an asynchronous method, a configuration error might exist. Check the system log file for configuration error messages.

Analyze the trace data or forward it to the appropriate organization for analysis. EJB asynchronous method scheduling and invocation are traced in the EJB container trace. For instructions on enabling this trace see the information about enabling trace on a running server. To analyze trace data, see information about trace output.

Configuring remote asynchronous EJB method results

Use this task to set the maximum number of unclaimed results for a remote asynchronous Enterprise JavaBeans (EJB) method call.

About this task

When a remote asynchronous EJB method is called, the server must save the results of the remote method invocation until the client claims the results using the Future.get method. If the client never claims the result, unclaimed results can accumulate in the server and use memory. To avoid using too much memory, the server limits the number of unclaimed results to 1000 by default. If the number of unclaimed results approaches or exceeds the limit, the server issues the `CNTR0328W` warning.

Procedure

1. Optional: Open the administrative console.
2. Select **Servers**.
3. Select **Server Types**.
4. Select **WebSphere application servers**.
5. Select the server that you want to configure.
6. From Server Infrastructure, select **Java and Process Management Process definition.**
7. From Additional Properties, select **Java Virtual Machine** .
8. In the Additional Properties area, select **Custom Properties** .
9. On the Application servers page, click **New** to specify an arbitrary name and value pair for your server.
10. In the **Name** entry field, type: `com.ibm.websphere.ejbcontainer.maxUnclaimedAsyncResults`
11. In the **Value** entry field, enter the wanted maximum number of unclaimed results. The special value `0` is interpreted as unlimited. The default value is `1000`.
12. Click **OK**.

13. Save the configuration.
14. Restart the server.

Results

The maximum number of unclaimed asynchronous EJB method results for all EJBs is set.

Configuring EJB asynchronous methods using scripting

Use **wsadmin** scripting to configure Enterprise JavaBeans (EJB) asynchronous methods.

Before you begin

You have working knowledge of Jacl or Jython and **wsadmin** scripting.

About this task

The behavior for EJB asynchronous methods is configured using the EJBAsync configuration object in the `server.xml` file. If you are using EJB asynchronous methods, then you might must update the EJBAsync configuration object to obtain the best settings for your environment. The EJBAsync configuration object exists at the server level. This means that each server in a multiple-server environment has its own EJBAsync configuration object and must be configured individually.

Procedure

1. Launch the **wsadmin** scripting tool using the Jython scripting language.
2. Determine the attributes on the EJBAsync configuration object that must be updated. You can update the following attributes on the EJBAsync configuration object:

Table 58. Attributes on EJBAsync configuration object. This table describes the attributes on the EJBAsync configuration object.

Attribute	Description
maxThreads	Specifies the maximum number of threads that are used in the execution of asynchronous EJB methods. The default value is 5.
workReqQSize	Specifies the size of the work request queue. The work request queue is a buffer that holds requested asynchronous methods until a thread is available to run them on. The sum of the maxThreads and the workReqQSize attributes is the total number of allowable in progress method requests. For example, if the maxThreads is set to 5 threads, and the workReqQSize is set to 50, then the total number of allowable in-progress method requests is 55. The default value is 0, indicating that the queue size is managed by the runtime environment. The run time currently uses the larger of 20 and maxThreads.
workReqQFullAction	Specifies the action taken when the thread pool is exhausted, and work request queue is full. If set to 1, an exception occurs instead of waiting for a thread, or a place in the queue, to become available. If set to 0, the thread that is requesting the asynchronous method execution waits until a thread, or place in the queue, becomes available. The default value is 0.
customWorkManagerJNDIName	Specifies the Java Naming and Directory Interface (JNDI) name used to look up the custom defined work manager in the namespace. The default value is null.

Table 58. Attributes on EJBAsync configuration object (continued). This table describes the attributes on the EJBAsync configuration object.

Attribute	Description
useCustomDefinedWM	Specifies whether a custom-defined work manager instance is used, or the default internal work manager instance. When the useCustomDefinedWM attribute is set to true, this means that a custom work manager instance is used. In this case, the customWorkManagerJNDIName attribute must be set, and all other attributes are ignored. When the useCustomDefinedWM attribute is set to `false`, the default, internal work manager instance is used. In this case, the customWorkManagerJNDIName attribute is ignored, and all other attributes are used to help configure the default work manager instance. The default value is false.
futureTimeout	Specifies the amount of time, in seconds, that the server-side future object, which is created as a result of running a fire-and-return asynchronous method, is available. The server-side future object is not valid after you call the get() method, and a value is returned to the remote client. To avoid memory leaks, you must call the get() method on the future object, or specify a positive and non-zero future duration value. A future duration value of zero indicates that the future object never times out. The default value is `86400`, which means that the future object expires and gets cleaned up by the application server after 24 hours and is no longer available. A org.omg.CORBA.OBJECT_NOT_EXIST exception is thrown when a call to the get() method is made after the future object expires. **Note:** This value is only applicable for clients that call the enterprise bean using a remote business interface; the value is not used for local business interface or no-interface views. When the asynchronous work has completed, the server sets an alarm for the duration specified to the server-side future object. When the alarm is activated, the server releases all the resources associated with the future object, making it unavailable to the client. If the client calls the get() method on the future object before the duration amount of time, the alarm is canceled and all the resources associated with the future object are released. **Note:** This attribute might affect the number of future objects on the server. Use the AsynchFutureObjectCount PMI statistic to determine the count of open FutureObjects on the server, which can help you determine whether applications are accumulating future objects without calling the get() method on those objects. See the topic, Enterprise bean counters, for more information.

3. Obtain a reference to the correct EJBAsync configuration object and store it in a variable.

   Using Jacl:

   ```
   set async [$AdminConfig list EJBAsync]
   ```

   Using Jython:

   ```
   async = AdminConfig.list('EJBAsync')
   ```

   If you have a multiple-server environment, then multiple EJBAsync configuration objects are returned. Programmatically loop over the list and select the EJBAsync configuration object that corresponds to the server you must update.

   In a multiple-server environment, as an alternative to programmatically looping over the list of EJBAsync objects, you can manually select the correct EJBAsync object and copy and paste it into your variable.

   For example, the output of the `AdminConfig list` command is:

   ```
   (cells/myNode04Cell/nodes/myCellManager01/servers/dmgr|server.xml#EJBAsync_1)(cells/
   myNode04Cell/nodes/myNode04/servers/server1|server.xml#EJBAsync_1247498700906)
   ```

   You can copy and paste the reference for the needed EJBAsync object into your variable.

   Using Jacl:

   ```
   set async "(cells/myNode04Cell/nodes/myNode04/servers/server1|server.xml#EJBAsync_1247498700906)"
   ```

   Using Jython:

   ```
   async = "(cells/myNode04Cell/nodes/myNode04/servers/server1|server.xml#EJBAsync_1247498700906)"
   ```

4. Update attributes on the EJBAsync configuration object.

   Update attributes on the EJBAsync configuration object using the **AdminConfig modify** command. As input to the command, specify the EJBAsync reference that you obtained in the previous step, and a list of attributeName and attributeValue combinations.

   To set a max thread count of `10` threads, a queue size of `15`, and a futureTimeout of `3600` seconds:

Using Jacl:

```
set update "{maxThreads 10} {workReqQSize 15} {futureTimeout 3600}"
$AdminConfig modify $async $update
```

Using Jython:

```
AdminConfig.modify(async, '[ [maxThreads "10"] [workReqQSize "15"] [futureTimeout "3600"] ]')
```

5. Save the configuration changes.

   Using Jython:

   ```
   AdminConfig.save()
   ```

   Using Jacl:

   ```
   $AdminConfig save
   ```

6. In a Network Deployment environment only, synchronize the node.

   Using Jacl:

   ```
   set sync1 [$AdminControl completeObjectName type=NodeSync,node=<your node>,*]
   $AdminControl invoke $sync1 sync
   ```

   Using Jython:

   ```
   sync1 = AdminControl.completeObjectName('type=NodeSync,node=<your node>,*')
   AdminControl.invoke(sync1, 'sync')
   ```

   You must run the node synchronization in these examples while connected to the server.

Results

As a result of your updates, the EJBAsync configuration object now reflects the attribute values that you specified. Restart your server so that the changes are updated on the server.

EJB 3.1 asynchronous methods

The Enterprise JavaBeans™ (EJB) 3.1 specification includes functionality that application developers can use to configure EJB asynchronous methods, which are run on a separate thread from the caller thread.

This mechanism decouples the client invocation request from the actual method execution. The client thread can continue doing other work while the EJB method is run on a separate thread, as directed by the EJB container.

Later, the client might want to examine the result of the asynchronous method execution, which is sometimes referred to as *fire and return*. In this case, the EJB container returns to the client an object that implements the java.util.concurrent.Future<V> interface. The client can use this object to check status, results, or exceptions from the asynchronous method invocation. Alternatively, asynchronous methods might not return any results, which is sometimes referred to as *fire and forget.*

For more details, see information about how to use EJB asynchronous methods in your application.

Here are some example usage scenarios for EJB asynchronous methods:

- An application has multiple, independent, pieces of work that all must be executed to produce a final result. For example, suppose that a travel reservation consists of three parts:
  1. Making a plane reservation.
  2. Making a rental car reservation.
  3. Making a hotel reservation.

  In this example, a client can use EJB asynchronous methods to process the reservation requests in parallel. After all three reservation methods run, the client aggregates the results into a complete travel reservation.
- An application has multiple, independent, pieces of work that it must run, and the application is not concerned about the results of this work. For example, suppose that a retailer has four branch stores, and the home office wants to print a sales report from each store when the business day ends. The

application developer can use EJB asynchronous methods as a batch processing mechanism. Multiple EJB method calls can be used to send a batch of get sales report requests, one to each branch store.

In this example, presumably the application does not need to check for results from these method calls. Perhaps this is handled by the home office employee who picks up the sales reports from a printer the next morning. Suppose that one of the branch stores failed to provide the requested report. The person collecting the reports can decide if that was expected, for example, the branch store was closed for renovations, or if the get sales report request must be reissued to that branch store.

Developing client code that calls EJB asynchronous methods

You can use the sample code within this topic to develop client code that calls EJB asynchronous methods.

Before you begin

This task assumes that the following interface and bean implementation classes exist:

```
public interface AcmeRemoteInterface {
   void fireAndForgetMethod ();
   Future<Integer> methodWithResults() throws AcmeException1, AcmeException2;
}
```

```
@Stateless
@Remote(AcmeRemoteInterface.class)
@Asynchronous
public class AcmeAsyncBean {
   public void fireAndForgetMethod () {
      // do non-critical work
   }

   public Integer methodWithResults() {
      Integer result;
      // do work, and then return results
      return result;
   }
}
```

About this task

Procedure

- Create client code that calls an asynchronous method where no results are returned, sometimes called a *fire and forget* method. This type of asynchronous method cannot result in application exceptions. However, system exceptions can occur that must be resolved.

  ```
  @EJB AcmeRemoteInterface myAsyncBean;

  try {
     myAsyncBean.fireAndForgetMethod();
  } catch (EJBException ejbex) {
     // Asynchronous method never dispatched, handle system error
  }
  ```

- Create client code that calls an asynchronous method where results are returned.
  - Create client code that calls an asynchronous method where the client waits for up to 5 seconds (the client thread is blocked during this time window) to receive results. Exception handling requirements are the same as in the previous step. For example:

    ```
    myResult = myFutureResult.get(5, TimeUnit.SECONDS);
    ```

  - Create client code that calls an asynchronous method where results are not immediately obtained. After the method has executed, the client retrieves the results. This scheme prevents the client thread from blocking, and the client is free to execute other work while polling for results. Exception

handling requirements are the same as in the previous steps. In this example, the client periodically polls the Future<V> object to determine when the asynchronous method has finished executing. For example:

```
while (!myFutureResult.isDone()) {
// Execute other work while waiting for the asynchronous method to complete.
}

// This call is guaranteed not to block because isDone returned true.
myResult = myFutureResult.get();
```

- Create client code to handle application exceptions. The client code calls an asynchronous method which returns an application exception in the Future<V> object. The following example demonstrates the exception handling code required to determine which application exception occurred.

```
@EJB AcmeRemoteInterface myAsyncBean;

Future<Integer>>myFutureResult = null;
Integer myResult = null;

try {
   myFutureResult = myAsyncBean.methodWithResults();
} catch (EJBException ejbx) {
   // Asynchronous method never dispatched, handle exception
}

// Method is eventually dispatched.  Wait for results.

try {
   myResult = myFutureResult.get();
} catch (ExecutionException ex) {
   // Determine which application exception that occurred during the
   // asynchronous method call.
   Throwable theCause = ex.getCause();
   if (theCause instanceof AcmeException1) {
      // Handle AcmeException1
   } else if (theCause instanceof AcmeException2) {
      // Handle AcmeException2
   } else {
      // Handle other causes.
   }
} catch ( ... ) {
   // Handle other exception.
}
```

- Create client code to identify system exceptions thrown by asynchronous method call during execution. The following example demonstrates the exception handling code required to determine if a system exception occurred.

```
@EJB AcmeRemoteInterface myAsyncBean;

Future<Integer>>myFutureResult = null;
Integer myResult = null;

try {
   myFutureResult = myAsyncBean.methodWithResults();
} catch (EJBException ejbx) {
   // Asynchronous method was not dispatched; handle exception.
}

// Method will eventually be dispatched so block now and wait for results

try {
   myResult = myFutureResult.get();
} catch (ExecutionException ex) {
   // Find the exception class that occurred during the asynchronous method
   Throwable theCause = ex.getCause();
   if (theCause instanceof EJBException) {
      // Handle the EJBException that might be wrapping a system exception
```

```
      // which occurred during the asynchronous method execution.
      Throwable theRootCause = theCause.getCause();
      if (theRootCause != null) {
         // Handle the system exception
      }
   } else ... // handle other causes
} catch (...) {
   // handle other exceptions
}
```

- Optionally create client code to cancel an asynchronous method call. If this attempt is successful, then the Future.isCancelled method returns true and the Future.get methods result in the CancellationException. The following example demonstrates the code required to cancel an asynchronous method call.

```
@EJB AcmeRemoteInterface myAsyncBean;

Future<Integer> myFutureResult = myFutureResult.methodWithResults();
Integer myResult;

if (myFutureResult.cancel(true)) {
   // Asynchronous method was not dispatched.
} else {
   // Asynchronous method already started executing.  The bean can still check
   // whether an attempt was made to cancel the call.
}

if (myFutureResult.isCancelled()) {
   // Asynchronous method call did not start executing because the cancel
   // method returned true in a previous line of the example.
}

try {
   myResult = myFutureResult.get();
} catch (CancellationException ex) {
   // Handle the exception that occurs because the cancel method returned true
   // in a previous line of the example.
}
```

- Optionally create bean code to check whether a client attempted to cancel the asynchronous method call. The following example demonstrates the bean code required to check whether the client attempted to cancel the asynchronous method call. If the client attempted to cancel the work, then the SessionContext.wasCancelCalled method returns true, and the bean code can avoid unnecessary work.

```
@Resource SessionContext myContext;

public Future<Integer> methodWithResults() {
   for (int i = 0; i < 3; i++) {
      // Do one piece of long-running work.

      // Before continuing, check whether the client attempted to cancel this work.
      if (myContext.wasCancelCalled()) {
         throw new AcmeCancelCalledException();
      }
   }

   // ...
}
```

- Pass back multiple output values from the asynchronous method invocation.

  In some cases, a method must pass back multiple pieces of data.

  One way to accomplish this task is by using pass-by-reference semantics. In this approach, an object is passed into the method as a parameter, updated by the method, and then the updated value is available to the client. This approach does work for asynchronous methods, but it is not the optimal pattern.

To return multiple pieces of data, create a wrapper inside the Future object that is returned by the method. In this approach, a wrapper object is defined that contains instance variables which hold the different pieces of data that must be returned. The asynchronous method sets these pieces of data into the wrapper object and returns it, and the client code then retrieves this data from the Future object.

Embedding multiple pieces of data inside the wrapper object is a local or remote, transparent pattern, that identifies exactly when the results are available. In contrast, the traditional pass-by-reference technique does not give the client an easy way to determine when the results are available. The passed in object is not updated until the asynchronous method runs, and the client cannot determine when that has occurred, other than by interrogating the Future object using the Future.isDone() or Future.get() methods.

```
// This is the result object that is returned from the asynchronous method.
// This object is wrappered in a Future object, and it contains the two pieces of data
// that must be returned from the method.
class ResultObject {
     public Boolean myResult;
     pubilc String myInfo;
}
// This is the asynchronous method code that gets the results and returns them.
@Asynchronous
public Future<ResultObject> asyncMethod1(Object someInputData) {
     boolean result = doSomeStuff();
     String info = doSomeMoreStuff();
     ResultObject theResult = new ResultObject();
     theResult.myResult = result;
     theResult.myInfo = info;
     return new javax.ejb.AsyncResult<ResultObject>(theResult);
}
// This is the client code that obtains the ResultObject, and then extracts the needed data from it.
Future<ResultObject> myFutureResult = myBeanRef.asyncMethod1(someInputData);
ResultObject theResult = myFutureResult.get();
boolean didItWork = theResult.myResult;
String explanation = theResult.myInfo;
```

Client programming model for EJB asynchronous methods

As documented in the Enterprise JavaBeans (EJB) 3.1 specification, you can invoke EJB asynchronous methods through the following interface types: local business, remote business, or no-interface view. Invocations through an EJB 2.1 client view, or a web services view, are not allowed.

The interface specification for an EJB asynchronous method must have a return type of void, or of type java.util.concurrent.Future <V>. No other return types are supported on the interface. As documented in the EJB 3.1 specification, the bean implementation method must have the same return type.

When your application does not need to examine the result of an EJB asynchronous method call, use an interface signature with a return type of void. Conversely, when your application needs to examine the result of an EJB asynchronous method call, use an interface with a return type of Future<V>.

In addition to considering whether results are examined, clients must be prepared to handle exceptions. As documented in the EJB 3.1 specification, the client receives an exception if the container is unable to allocate the internal resources required to schedule the asynchronous method for execution. In this case, the client can assume that the asynchronous method does not run. Also, exceptions can occur while the asynchronous method is running on the non-client thread.

Important: When an asynchronous method has a return type of void, the client has no mechanism to retrieve exception information. The EJB container logs an informational message in this case. However, for asynchronous methods that have a return type of Future<V>, the EJB container saves exception information in the Future<V> object. In this case, the get methods associated with the Future<V> object produce the exception, ExecutionException. The client must invoke the getCause method on the ExecutionException to retrieve details about the exception.

Clients must know that the get methods on the Future<V> object block the client thread if the asynchronous method has not finished running when the get method is called. If clients do not want this behavior, they can poll the Future<V> object to determine when the asynchronous method is finished by periodically calling the isDone method.

Finally, clients can use the Future<V> object to cancel an asynchronous method call. If you attempt to cancel an asynchronous method call while it is waiting to run, then it does not run, and other interactions with the Future<V> object reflect the cancellation. Otherwise, if you attempt to cancel an asynchronous method after it has started running, then it continues to run, but the bean method can still determine that the client attempted to cancel the call and respond with an application-specific return value or exception.

Another alternative is for the client to use the get method, which includes a timeout parameter. This get method only waits for results during the specified timeout period. The get method returns to the client as soon as the method has finished running, or when the timeout expires, even if the method has not finished running.

Nested asynchronous calls are supported; an asynchronous method call can be made from within an asynchronous method.

Note: When nesting asynchronous method calls, take into consideration the work manager settings to allow enough resources (maximum number of threads and work request queue size). For more information, read about the EJB container work manager for asynchronous methods.

Read about Developing client code that calls EJB asynchronous methods.

Bean implementation programming model for EJB asynchronous methods

You can configure asynchronous methods on session beans. However, as documented in the Enterprise JavaBeans (EJB) 3.1 specification, asynchronous methods must not be configured on entity beans, or message-driven beans.

Implementations of asynchronous methods must have a return type of void or of type, java.util.concurrent.Future <V>. No other return types are supported on the implementation. As documented in the EJB 3.1 specification, the bean implementation method must have the same return type as the interface specification. For more information, read about the client programming model for EJB asynchronous methods.

Nested asynchronous calls are supported; an asynchronous method call can be made from within an asynchronous method.

Note: When nesting asynchronous method calls, take into consideration the work manager settings to allow enough resources (maximum number of threads and work request queue size). For more information, read about the work manager for asynchronous methods.

Bean implementers must consider how long their asynchronous methods take to run because each request is invoked on a different thread. Another important consideration is the number of asynchronous method requests that an application is likely to start in parallel. These issues are important because the number of threads that are allocated to run asynchronous methods in the server process is a limited resource. Also, the buffer space that is allocated in the server process to queue asynchronous method requests, during times that all the allocated threads are busy, is a limited resource. For more information, read about the client programming model for EJB asynchronous methods.

Finally, bean implementers must follow the EJB 3.1 specification restrictions regarding the transaction attribute settings that are allowed on EJB asynchronous methods. For more information, read about the EJB container work manager for asynchronous methods.

EJB container work manager for asynchronous methods

The default work manager used by the Enterprise JavaBeans (EJB) container to dispatch asynchronous methods is not configurable. The following configuration information is provided to help you understand the limits imposed by this work manager. Remember, one EJB container work manager exists for all asynchronous methods running in the application server process.

The EJB container work manager has the following thread pool settings:

- Minimum number of threads = 1
- Maximum number of threads = 5
- Work request queue size = 0 work objects
- Action taken when buffer overflows = Block
- Remote Future object duration = 86400 seconds

Note: Work request queue size is the maximum number of asynchronous method requests that can be queued while waiting for a thread to become available. If you specify a value of 0 (the default) or blank is specified, the queue size is managed by the run time.

Additionally, the EJB container work manager has configuration settings that specify which service contexts are propagated from the client thread to the work manager thread that runs the asynchronous method. The global transaction service context and the activity session service context are not propagated. The security context, and all of the WebSphere extension contexts, such as work area, internationalization, and so on, are propagated for use on the execution thread.

EJB asynchronous methods settings

Use this page to modify settings on the work manager used in support of Enterprise JavaBeans asynchronous methods.

To view this administrative console page, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **EJB Container Settings** > **EJB asynchronous method invocation settings**

The Use internal work manager instance and Use custom work manager instance options are *mutually exclusive.*

Use this work manager for asynchronous methods

Specifies a default work manager instance.

The product provides an internal work manager instance for support of EJB asynchronous methods. The internal work manager instance is pre-configured for basic EJB asynchronous method functionality, and provides limited configuration settings.

Clicking this button specifies that you want to use the internal work manager instance to manage your asynchronous method invocations. Selecting this choice precludes the Use custom work manager instance option.

The Use internal work manager instance option is the default. Alternatively, you can use a custom-defined work manager instance.

Use custom work manager instance

Specifies a custom-defined work manager instance.

You can provide a more advanced configuration for EJB asynchronous methods by defining a custom work manager instance.

Selecting this choice allows you to use a work manager instance that you have already defined and configured in a way that is optimal for your environment.

Selecting this choice precludes the Use internal work manager instance option.

Maximum number of threads
Specifies the maximum number of threads that are used in the execution of asynchronous EJB methods. The default is 5.

Work request queue size
Specifies the size of the work request queue. The work request queue is a buffer that holds requested that asynchronous methods until a thread is available to run them on. The default is 0, indicating that the initial size is half way between the minimum and maximum number of threads.

The sum of the maximum number of threads and the work request queue size attributes is the total number of allowable in-progress method requests.

For example, if the maximum number of threads is set to five threads, and the work request queue size is set to 50, then the total number of allowable in-progress method requests is 55.

The default value is 0, indicating that the queue size is managed by the runtime environment. The runtime currently uses the larger of 20 and maxThreads.

Work request queue full action
Specifies the action taken when the thread pool is exhausted, and the work request queue is full. The default is BLOCK.

If set to FAIL, an exception occurs instead of waiting for a thread, or a place in the queue, to become available.

You might want to use this option during development to help identify errors caused by long-running asynchronous methods. You might also use it to help determine the number of threads and queue size that should be configured.

If set to BLOCK, the thread that is requesting that the asynchronous method execution waits until a thread, or a place in the queue, becomes available.

Work Manager JNDI name
Specifies the Java Naming and Directory Interface (JNDI) name used to look up the custom-defined work manager in the namespace.

Remote future object duration
Specifies the amount of time that the server retains the future object of each fire-and-return-results asynchronous method call. If an application does not retrieve the results within the specified period of time, the server purges the results of that method call to prevent memory leakage and a potential OutOfMemory exception. You can specify 0 to never purge the objects; however, specifying a zero value means that the future object never times out and you disable protection from incorrectly written programs that might cause the referenced OutOfMemory exception. The default value is 86400 seconds (24 hours).

Note: If you call the get() or the get(time) method on the future object, then you might decrease performance because blocking can occur until either the work is done or until the specified time has passed. Therefore, to avoid blocking on the get(..) methods, call the isDone() method, which returns immediately. Then, call the get() method after the isDone() method returns true.

Note: This value is only applicable for clients that call the enterprise bean using a remote business interface, the value is not used for local business interface or no-interface views. When the

asynchronous work has completed, the server sets an alarm for the duration specified to the server-side future object. When the alarm is activated, the server releases all the resources associated with the future object, making it unavailable to the client. If the client calls the get() method on the future object before the duration amount of time, the alarm is canceled and all the resources associated with the future object are released.

Developing session beans

Configuring EJB 3.1 session bean methods to be asynchronous

Use this task to configure Enterprise JavaBeans (EJB) 3.1 session bean methods to run asynchronously. You can make some or all of your bean methods asynchronous.

Before you begin

Attention: In EJB 3.1 modules, you can set one or more session bean methods to be asynchronous, broadening parallel processing in your application.

- If you are not already familiar with EJB 3.1 asynchronous methods, read about EJB 3.1 asynchronous methods, client programming model for EJB asynchronous methods, bean implementation programming model for EJB asynchronous methods, and EJB container work manager for asynchronous methods. The topics provide an overview of EJB 3.1 asynchronous methods, describe the client and bean implementation programming models, and discuss the work manager that the EJB container uses to dispatch asynchronous methods.
- Develop a new EJB 3.1 session bean for your application, or change an existing session bean so that it conforms to the EJB 3.1 programming model requirements for asynchronous methods. For general information, see information about developing enterprise beans.

About this task

After you have developed a session bean, complete the following steps to make one or more of the bean methods asynchronous.

Procedure

1. Specify one or more methods of the bean implementation class as asynchronous. This can be accomplished by adding @Asynchronous annotations in your bean source code, by adding <async-method> stanzas in your module deployment descriptor, or by adding a combination of both annotations and deployment descriptor stanzas. You can apply the @Asynchronous annotation or its superclasses only, to your bean implementation class. It cannot be applied to interface classes. Also, when the annotation is applied at the class level, all methods of that class are asynchronous. Likewise, all methods of a bean can be configured as asynchronous by applying "*" as the <method-name> in your deployment descriptor.

   See the following examples of applying the @Asynchronous annotation:

   - Apply the @Asynchronous annotation to one method of a bean with a no-interface view. In this example, the m1 method is synchronous and the m2 method is asynchronous.

```
@Stateless @LocalBean
public class MyLocalBean {

   public void m1() {

      // method code
   }

   @Asynchronous
   public Future<String> m2() {

      // method code
```

```
        return new javax.ejb.AsyncResult("Hello, Async World!");
    }
}
```

Important: The javax.ejb.AsyncResult<V> object is a convenience implementation of the Future<V> interface. See the API documentation for more details.

- Apply the @Asynchronous annotation to the class level of a bean class. In this example, both the m1 method and the m2 method are asynchronous on this no-interface view bean.

```
@Stateless @LocalBean @Asynchronouspublic class MyLocalBean {

    public void m1() {

        // method code
    }

    public Future<String> m2() {

        // method code

        return new javax.ejb.AsyncResult("Hello, Async World!");
    }

}
```

- Apply the @Asynchronous annotation to one method of a bean implementation class. In this example, the m1 method is synchronous and the m2 method is asynchronous. This example also demonstrates how the return types might differ between the business interface and the implementation class.

```
public interface MyIntf {

    public void m1();

    public Future<Integer> m2();

}
@Stateless @Local(MyIntf.class)
public class MyBean {

    public void m1() {

        // method code
    }

    @Asynchronous
    public Integer m2() {

        // method code

        return new Integer(3);
    }
}
```

- Apply the @Asynchronous annotation to the class level of a bean implementation class. In this example, both the m1 method and the m2 method are asynchronous.

```
@Stateless @Local(MyIntf.class) @Asynchronous
public class MyBean {

    public void m1() {

        // method code
    }

    public Integer m2() {
```

```
        // method code

        return new Integer(8);
    }
}
```

See the following examples of modifying the EJB module deployment descriptor, `ejb-jar.xml`:

- In this example all business methods of the FullAyncBean bean implementation class and its superclasses are configured as asynchronous with the wildcard (*) method-name element.

```
<session>
   <display-name>FullAsyncEJB</display-name>
   <ejb-name>FullAsyncBean</ejb-name>
   <business-local>com.ibm.sample.async.ejb.FullAsyncIntf</business-local>
   <ejb-class>com.ibm.sample.async.ejb.FullAsyncBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>*</method-name>
   </async-method>
</session>
```

- In this example only the specified methods and signatures -- all methods named m1 and the method m2 with a single String parameter -- are configured as asynchronous on the PartiallyAsyncBean bean implementation class.

```
<session>
   <display-name>PartiallyAsyncEJB</display-name>
   <ejb-name>PartiallyAsyncEJB</ejb-name>
   <business-local>com.ibm.sample.async.ejb.PartiallyAsyncIntf</business-local>
   <ejb-class>com.ibm.sample.async.ejb.PartiallyAsyncBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>m1</method-name>
   </async-method>
   <async-method>
      <method-name>m2</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </async-method>
</session>
```

See the following examples of applying a combination of the @Asynchronous annotation in the bean source code and modifying the EJB module deployment descriptor, `ejb-jar.xml`:

- In this example the @Asynchronous annotation configures method m2 to be asynchronous, and the deployment descriptor configures method m1 to also be an asynchronous method.

```
@Stateless @LocalBean
public class MyLocalBean {

   public void m1() {

      // method code
   }

   @Asynchronous
   public Future<String> m2() {

      // method code

      return new javax.ejb.AsyncResult("Hello, Async World!");
   }
}


<session>
   <display-name>MyLocalEJB</display-name>
   <ejb-name>MyLocalEJB</ejb-name>
   <local-bean/>
   <ejb-class>com.ibm.sample.async.ejb.MyLocalBean</ejb-class>
```

```
   <session-type>Stateless</session-type>   <async-method>
      <method-name>m1</method-name>
   </async-method>
</session>
```

- In this example the @Asynchronous annotation for method m2 is ignored because the deployment descriptor header contains the metadata-complete="true" flag. This flag causes configuration information to only be taken from the deployment descriptor elements. The result is that only method m1 of the MyLocalBean implementation is configured to be asynchronous.

```
@Stateless @LocalBean
public class MyLocalBean {

   public void m1() {

      // method code
   }

   @Asynchronous
   public Future<String> m2() {

      // method code

      return new javax.ejb.AsyncResult("Hello, Async World!");
   }
}


<ejb-jar id="ejb-jar_ID" ...
metadata-complete="true" version="3.1">
...
<session>
   <display-name>MyLocalEJB</display-name>
   <ejb-name>MyLocalEJB</ejb-name>
   <local-bean/>
   <ejb-class>com.ibm.sample.async.ejb.MyLocalBean</ejb-class>
   <session-type>Stateless</session-type>   <async-method>
      <method-name>m1</method-name>
   </async-method>
</session>
...
</ejb-jar>
```

2. Verify that the transaction attribute applied to any asynchronous method is either REQUIRED, REQUIRES_NEW, or NOT_SUPPORTED. These transaction attribute types are the only transaction attribute types supported on asynchronous methods. You can complete this action by either applying @TransactionAttribute annotations in the bean source code, by adding <container-transaction> stanzas in the `ejb-jar.xml` file, or by adding a combination of both annotations and <container-transaction> stanzas in the deployment descriptor.

   See the following example of setting the transaction attribute of an asynchronous method using annotations:

```
@Singleton @LocalBean
public class FullAsyncBean {
 @Asynchronous
 @TransactionAttribute(REQUIRED) // the default; specified for illustration
 public void m1() {
  // ...
 }

 @Asynchronous
 @TransactionAttribute(NOT_SUPPORTED)
 public void m2() {
  // ...
 }

 @Asynchronous
```

```
 @TransactionAttribute(REQUIRES_NEW)
 public void m3() {
  // ...
 }

 // ...
}
```

See the following example of setting the transaction attribute of an asynchronous method using the XML deployment descriptor:

```
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>FullAsyncBean</ejb-name>
        <method-name>m1</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>FullAsyncBean</ejb-name>
        <method-name>m2</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>FullAsyncBean</ejb-name>
        <method-name>m3</method-name>
      </method>
      <trans-attribute>RequiresNew</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
```

See the following example of using a combination of both annotations and the XML deployment descriptor to configure the transaction attributes of a bean. In this example the deployment descriptor stanzas for method m3 override the class level annotation. The result is that method m3 is configured as REQUIRES_NEW, while methods m1 and m2 are configured as REQUIRED:

```
@Singleton @LocalBean
@Asynchronous
@TransactionAttribute(REQUIRED) // the default; specified for illustration
public class FullAsyncBean {

 public void m1() {
  // ...
 }

 public void m2() {
  // ...
 }

 public void m3() {
  // ...
 }

 // ...
}


<assembly-descriptor>

   <container-transaction>
      <method>
         <ejb-name>FullAsyncBean</ejb-name>
         <method-name>m3</method-name>
```

```
        </method>
        <trans-attribute>RequiresNew</trans-attribute>
      </container-transaction>

  </assembly-descriptor>
```

What to do next

Continue to develop additional components for your application, or if you have finished all components required by your application, assemble and deploy your application. See information about assembling EJB modules and deploying EJB modules.

When you run your application, if it fails when it first attempts to use a session bean that has an asynchronous method, a configuration error might exist. Check the system log file for configuration error messages.

Analyze the trace data or forward it to the appropriate organization for analysis. EJB asynchronous method scheduling and invocation are traced in the EJB container trace. For instructions on enabling this trace see the information about enabling trace on a running server. To analyze trace data, see information about trace output.

Configuring remote asynchronous EJB method results

Use this task to set the maximum number of unclaimed results for a remote asynchronous Enterprise JavaBeans (EJB) method call.

About this task

When a remote asynchronous EJB method is called, the server must save the results of the remote method invocation until the client claims the results using the Future.get method. If the client never claims the result, unclaimed results can accumulate in the server and use memory. To avoid using too much memory, the server limits the number of unclaimed results to 1000 by default. If the number of unclaimed results approaches or exceeds the limit, the server issues the `CNTR0328W` warning.

Procedure

1. Optional: Open the administrative console.
2. Select **Servers**.
3. Select **Server Types**.
4. Select **WebSphere application servers**.
5. Select the server that you want to configure.
6. From Server Infrastructure, select **Java and Process Management Process definition.**
7. From Additional Properties, select **Java Virtual Machine** .
8. In the Additional Properties area, select **Custom Properties** .
9. On the Application servers page, click **New** to specify an arbitrary name and value pair for your server.
10. In the **Name** entry field, type: `com.ibm.websphere.ejbcontainer.maxUnclaimedAsyncResults`
11. In the **Value** entry field, enter the wanted maximum number of unclaimed results. The special value `0` is interpreted as unlimited. The default value is `1000`.
12. Click **OK**.
13. Save the configuration.
14. Restart the server.

Results

The maximum number of unclaimed asynchronous EJB method results for all EJBs is set.

Configuring EJB asynchronous methods using scripting

Use **wsadmin** scripting to configure Enterprise JavaBeans (EJB) asynchronous methods.

Before you begin

You have working knowledge of Jacl or Jython and **wsadmin** scripting.

About this task

The behavior for EJB asynchronous methods is configured using the EJBAsync configuration object in the `server.xml` file. If you are using EJB asynchronous methods, then you might must update the EJBAsync configuration object to obtain the best settings for your environment. The EJBAsync configuration object exists at the server level. This means that each server in a multiple-server environment has its own EJBAsync configuration object and must be configured individually.

Procedure

1. Launch the **wsadmin** scripting tool using the Jython scripting language.
2. Determine the attributes on the EJBAsync configuration object that must be updated. You can update the following attributes on the EJBAsync configuration object:

Table 59. Attributes on EJBAsync configuration object. This table describes the attributes on the EJBAsync configuration object.

Attribute	Description
maxThreads	Specifies the maximum number of threads that are used in the execution of asynchronous EJB methods. The default value is `5`.
workReqQSize	Specifies the size of the work request queue. The work request queue is a buffer that holds requested asynchronous methods until a thread is available to run them on. The sum of the maxThreads and the workReqQSize attributes is the total number of allowable in progress method requests. For example, if the maxThreads is set to `5` threads, and the workReqQSize is set to `50`, then the total number of allowable in-progress method requests is `55`. The default value is 0, indicating that the queue size is managed by the runtime environment. The run time currently uses the larger of 20 and maxThreads.
workReqQFullAction	Specifies the action taken when the thread pool is exhausted, and work request queue is full. If set to 1, an exception occurs instead of waiting for a thread, or a place in the queue, to become available. If set to 0, the thread that is requesting the asynchronous method execution waits until a thread, or place in the queue, becomes available. The default value is 0.
customWorkManagerJNDIName	Specifies the Java Naming and Directory Interface (JNDI) name used to look up the custom defined work manager in the namespace. The default value is null.
useCustomDefinedWM	Specifies whether a custom-defined work manager instance is used, or the default internal work manager instance. When the useCustomDefinedWM attribute is set to true, this means that a custom work manager instance is used. In this case, the customWorkManagerJNDIName attribute must be set, and all other attributes are ignored. When the useCustomDefinedWM attribute is set to `false`, the default, internal work manager instance is used. In this case, the customWorkManagerJNDIName attribute is ignored, and all other attributes are used to help configure the default work manager instance. The default value is false.

Table 59. Attributes on EJBAsync configuration object (continued). This table describes the attributes on the EJBAsync configuration object.

Attribute	Description
futureTimeout	Specifies the amount of time, in seconds, that the server-side future object, which is created as a result of running a fire-and-return asynchronous method, is available. The server-side future object is not valid after you call the get() method, and a value is returned to the remote client. To avoid memory leaks, you must call the get() method on the future object, or specify a positive and non-zero future duration value. A future duration value of zero indicates that the future object never times out. The default value is `86400`, which means that the future object expires and gets cleaned up by the application server after 24 hours and is no longer available. A org.omg.CORBA.OBJECT_NOT_EXIST exception is thrown when a call to the get() method is made after the future object expires. **Note:** This value is only applicable for clients that call the enterprise bean using a remote business interface; the value is not used for local business interface or no-interface views. When the asynchronous work has completed, the server sets an alarm for the duration specified to the server-side future object. When the alarm is activated, the server releases all the resources associated with the future object, making it unavailable to the client. If the client calls the get() method on the future object before the duration amount of time, the alarm is canceled and all the resources associated with the future object are released. **Note:** This attribute might affect the number of future objects on the server. Use the AsynchFutureObjectCount PMI statistic to determine the count of open FutureObjects on the server, which can help you determine whether applications are accumulating future objects without calling the get() method on those objects. See the topic, Enterprise bean counters, for more information.

3. Obtain a reference to the correct EJBAsync configuration object and store it in a variable.

   Using Jacl:

   ```
   set async [$AdminConfig list EJBAsync]
   ```

   Using Jython:

   ```
   async = AdminConfig.list('EJBAsync')
   ```

   If you have a multiple-server environment, then multiple EJBAsync configuration objects are returned. Programmatically loop over the list and select the EJBAsync configuration object that corresponds to the server you must update.

   In a multiple-server environment, as an alternative to programmatically looping over the list of EJBAsync objects, you can manually select the correct EJBAsync object and copy and paste it into your variable.

   For example, the output of the `AdminConfig list` command is:

   ```
   (cells/myNode04Cell/nodes/myCellManager01/servers/dmgr|server.xml#EJBAsync_1)(cells/
   myNode04Cell/nodes/myNode04/servers/server1|server.xml#EJBAsync_1247498700906)
   ```

   You can copy and paste the reference for the needed EJBAsync object into your variable.

   Using Jacl:

   ```
   set async "(cells/myNode04Cell/nodes/myNode04/servers/server1|server.xml#EJBAsync_1247498700906)"
   ```

   Using Jython:

   ```
   async = "(cells/myNode04Cell/nodes/myNode04/servers/server1|server.xml#EJBAsync_1247498700906)"
   ```

4. Update attributes on the EJBAsync configuration object.

   Update attributes on the EJBAsync configuration object using the **AdminConfig modify** command. As input to the command, specify the EJBAsync reference that you obtained in the previous step, and a list of attributeName and attributeValue combinations.

   To set a max thread count of `10` threads, a queue size of `15`, and a futureTimeout of `3600` seconds:

   Using Jacl:

   ```
   set update "{maxThreads 10} {workReqQSize 15} {futureTimeout 3600}"
   $AdminConfig modify $async $update
   ```

   Using Jython:

   ```
   AdminConfig.modify(async, '[ [maxThreads "10"] [workReqQSize "15"] [futureTimeout "3600"] ]')
   ```

5. Save the configuration changes.

   Using Jython:

```
AdminConfig.save()
```

Using Jacl:

```
$AdminConfig save
```

6. In a Network Deployment environment only, synchronize the node.

   Using Jacl:

   ```
   set sync1 [$AdminControl completeObjectName type=NodeSync,node=<your node>,*]
   $AdminControl invoke $sync1 sync
   ```

   Using Jython:

   ```
   sync1 = AdminControl.completeObjectName('type=NodeSync,node=<your node>,*')
   AdminControl.invoke(sync1, 'sync')
   ```

   You must run the node synchronization in these examples while connected to the server.

Results

As a result of your updates, the EJBAsync configuration object now reflects the attribute values that you specified. Restart your server so that the changes are updated on the server.

EJB 3.1 asynchronous methods

The Enterprise JavaBeans™ (EJB) 3.1 specification includes functionality that application developers can use to configure EJB asynchronous methods, which are run on a separate thread from the caller thread.

This mechanism decouples the client invocation request from the actual method execution. The client thread can continue doing other work while the EJB method is run on a separate thread, as directed by the EJB container.

Later, the client might want to examine the result of the asynchronous method execution, which is sometimes referred to as *fire and return*. In this case, the EJB container returns to the client an object that implements the java.util.concurrent.Future<V> interface. The client can use this object to check status, results, or exceptions from the asynchronous method invocation. Alternatively, asynchronous methods might not return any results, which is sometimes referred to as *fire and forget.*

For more details, see information about how to use EJB asynchronous methods in your application.

Here are some example usage scenarios for EJB asynchronous methods:

- An application has multiple, independent, pieces of work that all must be executed to produce a final result. For example, suppose that a travel reservation consists of three parts:
  1. Making a plane reservation.
  2. Making a rental car reservation.
  3. Making a hotel reservation.

  In this example, a client can use EJB asynchronous methods to process the reservation requests in parallel. After all three reservation methods run, the client aggregates the results into a complete travel reservation.
- An application has multiple, independent, pieces of work that it must run, and the application is not concerned about the results of this work. For example, suppose that a retailer has four branch stores, and the home office wants to print a sales report from each store when the business day ends. The application developer can use EJB asynchronous methods as a batch processing mechanism. Multiple EJB method calls can be used to send a batch of get sales report requests, one to each branch store.

  In this example, presumably the application does not need to check for results from these method calls. Perhaps this is handled by the home office employee who picks up the sales reports from a printer the next morning. Suppose that one of the branch stores failed to provide the requested report. The person collecting the reports can decide if that was expected, for example, the branch store was closed for renovations, or if the get sales report request must be reissued to that branch store.

Developing client code that calls EJB asynchronous methods

You can use the sample code within this topic to develop client code that calls EJB asynchronous methods.

Before you begin

This task assumes that the following interface and bean implementation classes exist:

```
public interface AcmeRemoteInterface {
   void fireAndForgetMethod ();
   Future<Integer> methodWithResults() throws AcmeException1, AcmeException2;
}
```

```
@Stateless
@Remote(AcmeRemoteInterface.class)
@Asynchronous
public class AcmeAsyncBean {
   public void fireAndForgetMethod () {
      // do non-critical work
   }

   public Integer methodWithResults() {
      Integer result;
      // do work, and then return results
      return result;
   }
}
```

About this task

Procedure

- Create client code that calls an asynchronous method where no results are returned, sometimes called a *fire and forget* method. This type of asynchronous method cannot result in application exceptions. However, system exceptions can occur that must be resolved.

  ```
  @EJB AcmeRemoteInterface myAsyncBean;

  try {
     myAsyncBean.fireAndForgetMethod();
  } catch (EJBException ejbex) {
     // Asynchronous method never dispatched, handle system error
  }
  ```

- Create client code that calls an asynchronous method where results are returned.
  - Create client code that calls an asynchronous method where the client waits for up to 5 seconds (the client thread is blocked during this time window) to receive results. Exception handling requirements are the same as in the previous step. For example:

    ```
    myResult = myFutureResult.get(5, TimeUnit.SECONDS);
    ```

  - Create client code that calls an asynchronous method where results are not immediately obtained. After the method has executed, the client retrieves the results. This scheme prevents the client thread from blocking, and the client is free to execute other work while polling for results. Exception handling requirements are the same as in the previous steps. In this example, the client periodically polls the Future<V> object to determine when the asynchronous method has finished executing. For example:

    ```
    while (!myFutureResult.isDone()) {
    // Execute other work while waiting for the asynchronous method to complete.
    }

    // This call is guaranteed not to block because isDone returned true.
    myResult = myFutureResult.get();
    ```

- Create client code to handle application exceptions. The client code calls an asynchronous method which returns an application exception in the Future<V> object. The following example demonstrates the exception handling code required to determine which application exception occurred.

```
@EJB AcmeRemoteInterface myAsyncBean;

Future<Integer>>myFutureResult = null;
Integer myResult = null;

try {
   myFutureResult = myAsyncBean.methodWithResults();
} catch (EJBException ejbx) {
   // Asynchronous method never dispatched, handle exception
}

// Method is eventually dispatched.  Wait for results.

try {
   myResult = myFutureResult.get();
} catch (ExecutionException ex) {
   // Determine which application exception that occurred during the
   // asynchronous method call.
   Throwable theCause = ex.getCause();
   if (theCause instanceof AcmeException1) {
      // Handle AcmeException1
   } else if (theCause instanceof AcmeException2) {
      // Handle AcmeException2
   } else {
      // Handle other causes.
   }
} catch ( ... ) {
   // Handle other exception.
}
```

- Create client code to identify system exceptions thrown by asynchronous method call during execution. The following example demonstrates the exception handling code required to determine if a system exception occurred.

```
@EJB AcmeRemoteInterface myAsyncBean;

Future<Integer>>myFutureResult = null;
Integer myResult = null;

try {
   myFutureResult = myAsyncBean.methodWithResults();
} catch (EJBException ejbx) {
   // Asynchronous method was not dispatched; handle exception.
}

// Method will eventually be dispatched so block now and wait for results

try {
   myResult = myFutureResult.get();
} catch (ExecutionException ex) {
   // Find the exception class that occurred during the asynchronous method
   Throwable theCause = ex.getCause();
   if (theCause instanceof EJBException) {
      // Handle the EJBException that might be wrapping a system exception
      // which occurred during the asynchronous method execution.
      Throwable theRootCause = theCause.getCause();
      if (theRootCause != null) {
         // Handle the system exception
      }
   } else ... // handle other causes
} catch (...) {
   // handle other exceptions
}
```

- Optionally create client code to cancel an asynchronous method call. If this attempt is successful, then the Future.isCancelled method returns true and the Future.get methods result in the CancellationException. The following example demonstrates the code required to cancel an asynchronous method call.

```
@EJB AcmeRemoteInterface myAsyncBean;

Future<Integer> myFutureResult = myFutureResult.methodWithResults();
Integer myResult;

if (myFutureResult.cancel(true)) {
   // Asynchronous method was not dispatched.
} else {
   // Asynchronous method already started executing.  The bean can still check
   // whether an attempt was made to cancel the call.
}

if (myFutureResult.isCancelled()) {
   // Asynchronous method call did not start executing because the cancel
   // method returned true in a previous line of the example.
}

try {
   myResult = myFutureResult.get();
} catch (CancellationException ex) {
   // Handle the exception that occurs because the cancel method returned true
   // in a previous line of the example.
}
```

- Optionally create bean code to check whether a client attempted to cancel the asynchronous method call. The following example demonstrates the bean code required to check whether the client attempted to cancel the asynchronous method call. If the client attempted to cancel the work, then the SessionContext.wasCancelCalled method returns true, and the bean code can avoid unnecessary work.

```
@Resource SessionContext myContext;

public Future<Integer> methodWithResults() {
   for (int i = 0; i < 3; i++) {
      // Do one piece of long-running work.

      // Before continuing, check whether the client attempted to cancel this work.
      if (myContext.wasCancelCalled()) {
         throw new AcmeCancelCalledException();
      }
   }

   // ...
}
```

- Pass back multiple output values from the asynchronous method invocation.

  In some cases, a method must pass back multiple pieces of data.

  One way to accomplish this task is by using pass-by-reference semantics. In this approach, an object is passed into the method as a parameter, updated by the method, and then the updated value is available to the client. This approach does work for asynchronous methods, but it is not the optimal pattern.

  To return multiple pieces of data, create a wrapper inside the Future object that is returned by the method. In this approach, a wrapper object is defined that contains instance variables which hold the different pieces of data that must be returned. The asynchronous method sets these pieces of data into the wrapper object and returns it, and the client code then retrieves this data from the Future object.

  Embedding multiple pieces of data inside the wrapper object is a local or remote, transparent pattern, that identifies exactly when the results are available. In contrast, the traditional pass-by-reference technique does not give the client an easy way to determine when the results are available. The passed

in object is not updated until the asynchronous method runs, and the client cannot determine when that has occurred, other than by interrogating the Future object using the Future.isDone() or Future.get() methods.

```
// This is the result object that is returned from the asynchronous method.
// This object is wrappered in a Future object, and it contains the two pieces of data
// that must be returned from the method.
class ResultObject {
     public Boolean myResult;
     pubilc String myInfo;
}
// This is the asynchronous method code that gets the results and returns them.
@Asynchronous
public Future<ResultObject> asyncMethod1(Object someInputData) {
     boolean result = doSomeStuff();
     String info = doSomeMoreStuff();
     ResultObject theResult = new ResultObject();
     theResult.myResult = result;
     theResult.myInfo = info;
     return new javax.ejb.AsyncResult<ResultObject>(theResult);
}
// This is the client code that obtains the ResultObject, and then extracts the needed data from it.
Future<ResultObject> myFutureResult = myBeanRef.asyncMethod1(someInputData);
ResultObject theResult = myFutureResult.get();
boolean didItWork = theResult.myResult;
String explanation = theResult.myInfo;
```

Client programming model for EJB asynchronous methods:

As documented in the Enterprise JavaBeans (EJB) 3.1 specification, you can invoke EJB asynchronous methods through the following interface types: local business, remote business, or no-interface view. Invocations through an EJB 2.1 client view, or a web services view, are not allowed.

The interface specification for an EJB asynchronous method must have a return type of void, or of type java.util.concurrent.Future <V>. No other return types are supported on the interface. As documented in the EJB 3.1 specification, the bean implementation method must have the same return type.

When your application does not need to examine the result of an EJB asynchronous method call, use an interface signature with a return type of void. Conversely, when your application needs to examine the result of an EJB asynchronous method call, use an interface with a return type of Future<V>.

In addition to considering whether results are examined, clients must be prepared to handle exceptions. As documented in the EJB 3.1 specification, the client receives an exception if the container is unable to allocate the internal resources required to schedule the asynchronous method for execution. In this case, the client can assume that the asynchronous method does not run. Also, exceptions can occur while the asynchronous method is running on the non-client thread.

Important: When an asynchronous method has a return type of void, the client has no mechanism to retrieve exception information. The EJB container logs an informational message in this case. However, for asynchronous methods that have a return type of Future<V>, the EJB container saves exception information in the Future<V> object. In this case, the get methods associated with the Future<V> object produce the exception, ExecutionException. The client must invoke the getCause method on the ExecutionException to retrieve details about the exception.

Clients must know that the get methods on the Future<V> object block the client thread if the asynchronous method has not finished running when the get method is called. If clients do not want this behavior, they can poll the Future<V> object to determine when the asynchronous method is finished by periodically calling the isDone method.

Finally, clients can use the Future<V> object to cancel an asynchronous method call. If you attempt to cancel an asynchronous method call while it is waiting to run, then it does not run, and other interactions with the Future<V> object reflect the cancellation. Otherwise, if you attempt to cancel an asynchronous method after it has started running, then it continues to run, but the bean method can still determine that the client attempted to cancel the call and respond with an application-specific return value or exception.

Another alternative is for the client to use the get method, which includes a timeout parameter. This get method only waits for results during the specified timeout period. The get method returns to the client as soon as the method has finished running, or when the timeout expires, even if the method has not finished running.

Nested asynchronous calls are supported; an asynchronous method call can be made from within an asynchronous method.

Note: When nesting asynchronous method calls, take into consideration the work manager settings to allow enough resources (maximum number of threads and work request queue size). For more information, read about the EJB container work manager for asynchronous methods.

Read about Developing client code that calls EJB asynchronous methods.

Bean implementation programming model for EJB asynchronous methods

You can configure asynchronous methods on session beans. However, as documented in the Enterprise JavaBeans (EJB) 3.1 specification, asynchronous methods must not be configured on entity beans, or message-driven beans.

Implementations of asynchronous methods must have a return type of void or of type, java.util.concurrent.Future <V>. No other return types are supported on the implementation. As documented in the EJB 3.1 specification, the bean implementation method must have the same return type as the interface specification. For more information, read about the client programming model for EJB asynchronous methods.

Nested asynchronous calls are supported; an asynchronous method call can be made from within an asynchronous method.

Note: When nesting asynchronous method calls, take into consideration the work manager settings to allow enough resources (maximum number of threads and work request queue size). For more information, read about the work manager for asynchronous methods.

Bean implementers must consider how long their asynchronous methods take to run because each request is invoked on a different thread. Another important consideration is the number of asynchronous method requests that an application is likely to start in parallel. These issues are important because the number of threads that are allocated to run asynchronous methods in the server process is a limited resource. Also, the buffer space that is allocated in the server process to queue asynchronous method requests, during times that all the allocated threads are busy, is a limited resource. For more information, read about the client programming model for EJB asynchronous methods.

Finally, bean implementers must follow the EJB 3.1 specification restrictions regarding the transaction attribute settings that are allowed on EJB asynchronous methods. For more information, read about the EJB container work manager for asynchronous methods.

EJB container work manager for asynchronous methods

The default work manager used by the Enterprise JavaBeans (EJB) container to dispatch asynchronous methods is not configurable. The following configuration information is provided to help you understand the limits imposed by this work manager. Remember, one EJB container work manager exists for all asynchronous methods running in the application server process.

The EJB container work manager has the following thread pool settings:

- Minimum number of threads = 1
- Maximum number of threads = 5
- Work request queue size = 0 work objects
- Action taken when buffer overflows = Block
- Remote Future object duration = 86400 seconds

Note: Work request queue size is the maximum number of asynchronous method requests that can be queued while waiting for a thread to become available. If you specify a value of 0 (the default) or blank is specified, the queue size is managed by the run time.

Additionally, the EJB container work manager has configuration settings that specify which service contexts are propagated from the client thread to the work manager thread that runs the asynchronous method. The global transaction service context and the activity session service context are not propagated. The security context, and all of the WebSphere extension contexts, such as work area, internationalization, and so on, are propagated for use on the execution thread.

EJB asynchronous methods settings

Use this page to modify settings on the work manager used in support of Enterprise JavaBeans asynchronous methods.

To view this administrative console page, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **EJB Container Settings** > **EJB asynchronous method invocation settings**

The Use internal work manager instance and Use custom work manager instance options are *mutually exclusive.*

Use this work manager for asynchronous methods:

Specifies a default work manager instance.

The product provides an internal work manager instance for support of EJB asynchronous methods. The internal work manager instance is pre-configured for basic EJB asynchronous method functionality, and provides limited configuration settings.

Clicking this button specifies that you want to use the internal work manager instance to manage your asynchronous method invocations. Selecting this choice precludes the Use custom work manager instance option.

The Use internal work manager instance option is the default. Alternatively, you can use a custom-defined work manager instance.

Use custom work manager instance:

Specifies a custom-defined work manager instance.

You can provide a more advanced configuration for EJB asynchronous methods by defining a custom work manager instance.

Selecting this choice allows you to use a work manager instance that you have already defined and configured in a way that is optimal for your environment.

Selecting this choice precludes the Use internal work manager instance option.

Maximum number of threads:

Specifies the maximum number of threads that are used in the execution of asynchronous EJB methods. The default is 5.

Work request queue size:

Specifies the size of the work request queue. The work request queue is a buffer that holds requested that asynchronous methods until a thread is available to run them on. The default is 0, indicating that the initial size is half way between the minimum and maximum number of threads.

The sum of the maximum number of threads and the work request queue size attributes is the total number of allowable in-progress method requests.

For example, if the maximum number of threads is set to five threads, and the work request queue size is set to 50, then the total number of allowable in-progress method requests is 55.

The default value is 0, indicating that the queue size is managed by the runtime environment. The runtime currently uses the larger of 20 and maxThreads.

Work request queue full action:

Specifies the action taken when the thread pool is exhausted, and the work request queue is full. The default is BLOCK.

If set to FAIL, an exception occurs instead of waiting for a thread, or a place in the queue, to become available.

You might want to use this option during development to help identify errors caused by long-running asynchronous methods. You might also use it to help determine the number of threads and queue size that should be configured.

If set to BLOCK, the thread that is requesting that the asynchronous method execution waits until a thread, or a place in the queue, becomes available.

Work Manager JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name used to look up the custom-defined work manager in the namespace.

Remote future object duration:

Specifies the amount of time that the server retains the future object of each fire-and-return-results asynchronous method call. If an application does not retrieve the results within the specified period of time, the server purges the results of that method call to prevent memory leakage and a potential OutOfMemory exception. You can specify 0 to never purge the objects; however, specifying a zero value means that the future object never times out and you disable protection from incorrectly written programs that might cause the referenced OutOfMemory exception. The default value is 86400 seconds (24 hours).

Note: If you call the get() or the get(time) method on the future object, then you might decrease performance because blocking can occur until either the work is done or until the specified time has passed. Therefore, to avoid blocking on the get(..) methods, call the isDone() method, which returns immediately. Then, call the get() method after the isDone() method returns true.

Note: This value is only applicable for clients that call the enterprise bean using a remote business interface, the value is not used for local business interface or no-interface views. When the asynchronous work has completed, the server sets an alarm for the duration specified to the server-side future object. When the alarm is activated, the server releases all the resources

associated with the future object, making it unavailable to the client. If the client calls the get() method on the future object before the duration amount of time, the alarm is canceled and all the resources associated with the future object are released.

Developing stateful session beans

You can create a bean implementation class for a stateful session bean as introduced in the Enterprise JavaBeans™ (EJB) 1.0 specification and significantly simplified by the EJB 3.0 specification. A stateful bean is a type of session bean that is intended for use by a single client during its lifetime and maintains a conversational state with the client that is calling it.

Before you begin

Make sure that you understand the inheritance rules for each annotation you implement. For example, the @TransactionManagement annotation is coded on the stateful session bean class only. You cannot use the @TransactionManagement annotation in the class that it extends, or any class higher in the class inheritance tree.

About this task

Stateful session beans can have the following views: no-interface local view (new in EJB 3.1), business local, business remote, EJB 2.1 local, and EJB2.1 remote client views. One example is a shopping cart where the client adds items to the cart over the course of an on-line shopping session.

The following example shows a basic stateful session bean:

```
package com.ibm.example;

public interface ShoppingCart {
    void addToCart (Object o);
    Collection getContents();
}

package com.ibm.example;

@Stateful
public class ShoppingCartBean implements ShoppingCart {
    private ArrayList contents = new ArrayList();

    public void addToCart (Object o) {
        contents.add(o);
    }
    public Collection getContents() {
        return contents;
    }
}
```

As with other enterprise bean types, you can also declare metadata for stateful session beans in the deployment descriptor rather than using annotations; for example:

```
<?xml version="1.0"?>
<ejb-jar
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1">
  <enterprise-beans>
    <ejb-name>ShoppingCartBean</ejb-name>
    <business-local>com.ibm.example.ShoppingCart</business-local>
    <ejb-class>com.ibm.example.ShoppingCartBean</ejb-class>
    <session-type>Stateful</session-type>
  </enterprise-beans>
</ejb-jar>
```

Procedure

- Code the initialization and destruction methods, understanding that they run in an unspecified security context and an unspecified transaction context. During initialization, the instance is created, dependency injection occurs, and PostConstruct life cycle interceptor callbacks are invoked. The PreDestroy life cycle interceptor callbacks are invoked for a stateful session bean when a remove method is called. Also keep in mind that the PreDestroy life cycle interceptor callbacks are not called if the stateful session bean times out while in the passive state, or if an unexpected exception occurs during a method invocation on the bean and the bean is discarded.
- Use PrePassivate and PostActivate methods if the stateful session bean can contain state that is not serializable. The container can passivate a stateful session bean instance anytime that it is not enlisted in a transaction or currently running a method request. The stateful session bean instance is moved to the passive state by serializing all the state data. If any of the state data does not serialize, the stateful session bean instance is discarded by the container.
- Consider implementing the optional javax.ejb.SessionSynchronization interface if the state data of the stateful session bean needs to be reset after a transaction rollback. The state data of a stateful session bean is not transactional and is not automatically reset to the initial state as the result of a transaction rollback. By implementing the afterCompletion method of the javax.ejb.SessionSynchronization interface, the stateful session bean instance might reset itself to the initial or consistent state.
- Use the @AccessTimeout notation to prohibit concurrent client requests or limit how long a method waits for the instance lock to be granted. By default, the container allows concurrent client requests, but serializes all method calls and container-invoked callbacks to prevent multi-threaded access to the stateful session bean instance. This behavior is like using container-managed concurrency with write locks for singleton session beans. However, unlike singleton session beans, stateful session beans cannot be configured to use bean-managed concurrency and the lock type cannot be changed. Only the access-timeout value can be modified for stateful session beans. The following code example illustrates a stateful session bean with a concurrent access-timeout value that prohibits concurrent client requests:

```
package com.ibm.example;
@Stateful
public class ShoppingCartBean implements ShoppingCart {
    private ArrayList contents = new ArrayList();

    @AccessTimeout( value=0 )
    public void addToCart (Object o) {
                contents.add(o);
    }
    public Collection getContents() {
                return contents;
    }
}
 package com.ibm.example;
@Stateful
public class ShoppingCartBean implements ShoppingCart {
    private ArrayList contents = new ArrayList();

    @AccessTimeout( value=0 )
    public void addToCart (Object o) {
                contents.add(o);
    }
    public Collection getContents() {
                return contents;
    }
}
```

  If no annotation is provided, the default behavior is to wait until a lock is granted. No time limit exists for how long a client waits for the lock to be granted. Because nothing is coded at the class level, no wait time limit exists for a lock to be granted for all methods of the class. If the @AccessTimeout annotation is used and the container cannot grant the lock within the specified time limit, a javax.ejb.ConcurrentAccessTimeoutException exception occurs on the client. The @AccessTimeout annotation only applies to methods that are declared in the same class as the @AccessTimeout annotation. For a given class, the metadata for the @AccessTimeout annotation is never inherited from a class higher in the class inheritance tree.

An access-timeout value of -1 indicates that concurrent method calls block access to the bean instance indefinitely (the default). An access-timeout value of 0 indicates that concurrent method calls are not allowed. The exception, javax.ejb.ConcurrentAccessException, occurs when concurrency is detected. And any positive value indicates the amount of time to wait until the method can proceed.

Prior to Java EE 6, the only concurrency behavior supported for stateful session beans was an access-timeout of -1, no concurrency. Since the Java EE 6 specification changed the default behavior, a system property is supported that provides the older default behavior. See EJB container system properties for more information about the com.ibm.websphere.ejbcontainer.EE5Compatibility system property.

You can also specify the @AccessTimeout annotation using the XML deployment descriptor, and if you use the XML deployment descriptor, the metadata from the @AccessTimeout annotation is ignored. The following example uses the XML deployment descriptor to specify the same metadata as the previous example.

```
<session>
   <ejb-name>ShoppingCartBean</ejb-name>
   <concurrent-method>
     <method>
       <method-name>addToCart</method-name>
     </method>
     <access-timeout>
       <timeout>0</timeout>
       <unit>Milliseconds</unit>
     </access-timeout>
   </concurrent-method>
 </session>
```

- It is important to know that the XML coding of the concurrent-methodType follows the three styles outlined in the EJB specification for composing the XML for container-transaction method elements. The three styles are: Style 1 uses the special method name * to apply the access-timeout value to all business methods for the specified bean.

```
<!-- Example: Style 1 -->
      <concurrent-method>
        <method>
          <method-name>*</method-name>
        </method>
       <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
```

Style 2 is used to refer to a business method with a specific name and assign it the specified access-timeout value. If the method name is overloaded, meaning multiple methods have the same name but different method signatures, all methods with this name have the specified access-timeout value.

```
<!-- Example: Style 2 -->
      <concurrent-method>
       <method>
          <method-name>businessMethod</method-name>
        </method>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
```

Style 3 is used to refer to a distinct method that matches the given method name and has a method signature that matches the method parameters listed. Style 3 takes precedence over both style 1 and style 2.

```
<!-- Example: Style 3 -->
      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
```

```
      <method-params>
        <method-param>long</method-param>
        <method-param>int</method-param>
      </method-params>
    </method>
    <access-timeout>
      <timeout>2000</timeout>
      <unit>Milliseconds</unit>
    </access-timeout>
  </concurrent-method>
```

If style 1 XML is used to define an access-timeout value, then all @AccessTimeout annotations on the bean are ignored because the method level one is used instead.

- Ensure that you understand the inheritance rules for the annotations that you have used.
- Stateful session beans are not reentrant, which means that the stateful session bean method calls itself back. If a reentrant call is made to a stateful session bean, a javax.ejb.ConcurrentAccessTimeoutException exception occurs on the client, regardless of the access-timeout value. This exception does not result in the stateful session bean being discarded, nor does it mark the transaction for rollback, unless the exception is not handled by the caller.

Developing a session bean to have a No-Interface Local view

You can specify that a session bean have a No-Interface view.

About this task

A session bean has a No-Interface Local view when:

- The bean does not expose any other client views (Local, Remote, 2.x Remote Home, 2.x Local Home, Web Service) and its implements clause is empty.
- The bean exposes at least one other client view. The bean designates that it exposes a no-interface view with the @LocalBean annotation on the bean class or in the deployment descriptor.

You can also declare metadata for a session bean with No-Interface Local view in the deployment descriptor rather than using annotations.

The following steps contain code snippets that demonstrate the code to use for a No-Interface view.

Procedure

- Specify that a session bean have No-Interface view because there are no declared interfaces.

```
@Stateless
public class CartBean
```

- Specify that a session bean have No-Interface view using the `@LocalBean` annotation.

```
@Stateless
@LocalBean
@Remote( Cart.class )
public class CartBean implements Cart
```

- Specify metadata for a session bean with No-Interface Local view in the deployment descriptor.

```
<?xml version="1.0"?>
<ejb-jar
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1">
  <enterprise-beans>
    <session>
      <ejb-name>CartBean</ejb-name>
      <local-bean/>
      <business-remote>com.ibm.example.Cart</business-remote>
      <ejb-class>com.ibm.example.CartBean</ejb-class>
```

```
        <session-type>Stateless</session-type>
      </session>
    </enterprise-beans>
  </ejb-jar>
```

No-Interface Local View

New in Enterprise JavaBeans (EJB) 3.1, session beans might now be exposed to clients through a No-Interface view. In prior versions of the specification, bean developers were required to provide an interface that would be used to expose the bean methods to a client.

The new No-Interface Local View enables the customer to use the Enterprise JavaBeans (EJB) class as the local interface. This is supported when:

- The bean does not expose any other client views (Local, Remote, 2.x Remote Home, 2.x Local Home, Web Service) and its implements clause is empty.
- The bean exposes at least one other client view. The bean designates that it exposes a no-interface view by means of the @LocalBean annotation on the bean class or in the deployment descriptor.

A session bean might now subclass another session bean

New in Enterprise JavaBeans (EJB) 3.1, a session bean might now subclass another session bean. Prior versions of the EJB specification restricted a session bean from inheriting from another session bean.

Important: This new support only enables implementation inheritance but does not provide component inheritance. Therefore, any meta data or annotations on the superclass that defined it to be an enterprise bean, will not be inherited by the subclass.

Developing singleton session beans

Create a bean implementation class for a singleton session bean, introduced by the Enterprise JavaBeans (EJB) 3.1 specification. The EJB container initializes only one instance of a singleton session bean, and that instance is shared by all clients. Because a single instance is shared by all clients, singleton session beans have special life cycle and concurrency semantics.

Before you begin

Make sure that you understand the inheritance rules for each annotation you implement. For example, the @ConcurrencyManagement annotation is coded on the singleton session bean class only. You cannot use the @ConcurrencyManagement annotation in the class that it extends, or any class higher in the class inheritance tree.

About this task

Singleton session beans can have business local, business remote, and web service client views; they cannot have EJB 2.1 local or remote client views. This singleton session bean support replaces the proprietary startup bean functionality, which has been deprecated.

The following example shows a basic singleton session bean:

```
public interface Configuration {
 Object get(String name);
 void set(String name, Object value);
}

@Singleton
public class ConfigurationBean implements Configuration {
 private Map<String, Object> settings = new HashMap<String, Object>();

 public Object get(String name) {
  return settings.get(name);
 }
```

```
 public void set(String name, Object value) {
  settings.put(name,value);
 }
}
```

As with other enterprise bean types, you can also declare metadata for singleton session beans in the deployment descriptor rather than using annotations; for example:

```
<?xml version="1.0"?>
<ejb-jar
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
  version="3.1"
>
  <enterprise-beans>
    <ejb-name>ConfigurationBean</ejb-name>
    <business-local>com.ibm.example.Configuration</business-local>
    <ejb-class>com.ibm.example.ConfigurationBean</ejb-class>
    <session-type>Singleton</session-type>
  </enterprise-beans>
</ejb-jar>
```

Procedure

- Code the initialization and destruction methods, understanding how they relate to your options for setting the transaction context. During initialization, the instance is created, dependency injection occurs, and PostConstruct life cycle interceptor callbacks are started. The PreDestroy life cycle interceptor callbacks are started for a singleton session bean when its containing application is stopped.

  It can be useful to complete transactional activities during the PostConstruct and PreDestroy life cycle interceptor callbacks. For this reason, singleton session bean life cycle interceptor callbacks have a well-defined transaction context. The following transaction context values are similar to @Timeout methods: only REQUIRED (default), REQUIRES_NEW, and NOT_SUPPORTED can be used, and REQUIRED is translated to REQUIRES_NEW.

  Transaction attributes are only recognized when they are specified on life cycle interceptor methods on the bean class. The same transaction context is used for all life cycle interceptors. The following example illustrates a singleton session bean with transaction attributes specified for the PostConstruct and PreDestroy life cycle interceptor callbacks.

  ```
  @Singleton
  public class ConfigurationBean implements Configuration {
   @PostConstruct
   @TransactionAttribute(REQUIRED) // the default; specified for illustration
   public void initialize() {
    // ...
   }

   @PreDestroy
   @TransactionAttribute(NOT_SUPPORTED)
   public void destroy() {
    // ...
   }

   // ...
  }
  ```

  Instead of using an annotation, you can specify the same metadata using the XML deployment descriptor. If you specify transaction attributes in the XML deployment descriptor, then any metadata obtained from the @TransactionAttribute annotation is ignored. The following example uses the XML deployment descriptor to specify the same metadata as the previous example.

  ```
    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>ConfigurationBean</ejb-name>
  ```

```
        <method-name>initialize</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>ConfigurationBean</ejb-name>
        <method-name>destroy</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
```

- Unless otherwise specified, the singleton session bean instance is typically initialized when the bean is first used through one of its client views, which is the same as any other session bean. Use the @Startup annotation or the corresponding XML deployment descriptor to mark a bean as a startup bean. Marking a singleton bean as a startup bean means that the EJB container must run the PostConstruct method before it supports any external client requests made to the application to run. A PostConstruct method in a singleton bean can create an EJB timer, add a message to a JMS queue or topic, call an asynchronous EJB method, or initiate other asynchronous mechanisms that call an EJB. However, to avoid a deadlock, the PostConstruct method must not wait for an EJB timer to run, a message-driven bean method to be called, or an asynchronous EJB method to finish.

  Application developers can place business logic in the PostConstruct methods of these startup singleton instances to complete tasks that must be performed before any client work being started by the container, such as preloading caches or initiating asynchronous work within the application.

  The following example illustrates a singleton session bean with startup initialization:

```
@Singleton
@Startup
public class ConfigurationBean implements Configuration {
 @PostConstruct
 public void initialize() {
  // 1. Create the database table if it does not exist.
  // 2. Initialize settings from the database table.
  // 3. Load a cache.
  // 4. Initiate asynchronous work (for example, work to a messaging queue or to
  //    calls to asynchronous session bean methods.
 }

 // ...
}
```

  Instead of using an annotation, you can specify the same metadata using the XML deployment descriptor. Specify `true` to mark this singleton bean as a startup singleton. Conversely, specify `false`, and the @Startup annotation is overridden, if it exists on the class file.

```
    <session>
      <ejb-name>ConfigurationBean</ejb-name>
      <init-on-startup>true</init-on-startup>
    </session>
```

- Determine whether the initialization method of the singleton session bean has an implicit dependency on another singleton session bean.

  If an implicit dependency exists, use dependency metadata to make the dependency explicit. The container ensures that dependency singleton beans are initialized before their dependent beans are initialized and that they are destroyed after their dependent beans are destroyed. The following example illustrates a singleton session bean with dependency metadata:

```
@Singleton
public class DatabaseBean {
 @PostConstruct
 public void initialize() {
  // Create database tables.
 }
}
```

```
@Singleton
@DependsOn({"DatabaseBean"})
public class ConfigurationBean implements Configuration {
 @PostConstruct
 public void initialize() {
  // Initialize settings from a database table.
 }

 // ...
}
```

Additionally, you can make cross-module dependencies by using the ejb-link `module.jar#bean` syntax. Circular dependencies are not supported, and cause an application to fail.

Instead of using an annotation, you can specify the same metadata using the XML deployment descriptor. If you specify dependency metadata in the XML deployment descriptor, then any metadata from the @DependsOn annotation is ignored.

```
<session>
  <ejb-name>ConfigurationBean</ejb-name>
  <depends-on>
    <ejb-name>DatabaseBean</ejb-name>
  </depends-on>
</session>
```

- Decide whether to use container-managed concurrency or bean-managed concurrency. The @Lock and @AccessTimeout annotations are not applicable when bean-managed concurrency is used.

  You can implement the @ConcurrencyManagement annotation on the singleton session bean class only. It cannot be used on the class it extends or any class higher in the class inheritance tree. The following code example illustrates a singleton with bean-managed concurrency:

```
@Singleton
@ConcurrencyManagement(BEAN)
public class ConfigurationBean implements Configuration {
 private Map<String, Object> settings = new HashMap<String, Object>();

 synchronized public Object get(String name) {
  return settings.get(name);
 }

 synchronized public void set(String name, Object value) {
  settings.put(name, value);
 }
}
```

  Instead of using an annotation, you can specify the same metadata using the XML deployment descriptor. If the metadata is specified in both the XML deployment descriptor and using the @ConcurrencyManagement annotation, then the value must match or the application fails. The following example uses the XML deployment descriptor to specify the same metadata as the previous example.

```
<session>
  <ejb-name>ConfigurationBean</ejb-name>
  <concurrency-management-type>Bean</concurrency-management-type>
</session>
```

  The container does not do locking for each method called. Instead, the actual bean is responsible for the locking that is required. In the example, the bean provider has chosen to implement the methods using the synchronized keyword. This is supported for singleton session beans with bean-managed concurrency, but it is not supported for other EJB component types. It is not required for the bean provider to use the synchronized keyword to provide concurrency. For example, the bean provider can use the java.util.concurrent.locks.ReentrantReadWriteLock class that is found in JDK 5 and later.

  The locking semantics required by the EJB 3.1 specification for container-managed concurrency matches the behavior of the java.util.concurrent.locks.ReentrantReadWriteLock class.

- If you are using container-managed concurrency, use the @Lock notation to manage concurrency of methods.

The following code example illustrates singleton with container-managed concurrency.

```
@Singleton
public class ConfigurationBean implements Configuration {
 private Map<String, Object> settings = new HashMap<String, Object>();

 @Lock(READ)
 public Object get(String name) {
  return settings.get(name);
 }

 public void set(String name, Object value) {
  settings.put(name, value);
 }
}
```

Instead of using an annotation, you can specify the same metadata using the XML deployment descriptor. If the metadata is specified in both the XML deployment descriptor and the @Lock annotation, then the metadata obtained from the @Lock annotation is ignored. The following example uses the XML deployment descriptor to specify the same metadata as the previous example.

```
    <session>
      <ejb-name>ConfigurationBean</ejb-name>
      <concurrent-method>
        <method>
          <method-name>get</method-name>
        </method>
        <lock>Read</lock>
      </concurrent-method>
    </session>
```

The example also illustrates annotating a method with @Lock(READ) to indicate that the container must get a read lock when that method is started. When a method is annotated with @Lock, it overrides the @Lock annotation that is specified at the class level. When there is no @Lock annotation at class level, the default is a write lock. In the example, the @Lock(READ) on the method is overriding the default write lock at the class level. When a method is not annotated at the method level and there is no annotation at the class level, the default of write lock is used by the container.

Because most methods require a read lock, use @Lock(READ) at the class level to indicate that all business methods in this class require the container to obtain a read lock. For methods that require a write lock, annotate those methods with @Lock(WRITE) to show that it overrides the read lock that was specified at the class level.

The following example illustrates this technique:

```
@Singleton
@Lock(READ)
public class ConfigurationBean implements Configuration {
 private Map<String, Object> settings = new HashMap<String, Object>();

 public Object get(String name) {
  return settings.get(name);
 }

 @Lock(WRITE)
 public void set(String name, Object value) {
  settings.put(name, value);
 }
}
```

The @Lock annotation applies only to methods that are declared in the same class as the @Lock annotation. For a given class, the metadata for @Lock is never inherited from a class higher in the class inheritance tree. Instead of using an annotation at the class level, the same metadata can be specified in the XML deployment descriptor by using the special method-name *, which matches all methods.

- If you are using container-managed concurrency, use the @AccessTimeout notation to limit how long a method waits for a lock to be granted. The following code example illustrates concurrent access timeouts.

```
@Singleton
public class ConfigurationBean implements Configuration {
 @Lock(READ)
 @AccessTimeout(1000)
 public Object get(String name) {
  // query the database
 }

 public void set(String name, Object value) {
  // update the database
 }
}
```

  If no annotation is provided, the method defaults to wait until a lock is granted. There is no time limit for how long a client waits for the lock to be granted. Because nothing is coded at the class level, there is no wait time limit for a lock to be granted for all methods of the class. If the @AccessTimeout annotation is used and the container cannot grant the lock within the specified time limit, a javax.ejb.ConcurrentAccessTimeoutException is thrown to the client. The @AccessTimeout annotation applies only to methods that are declared in the same class as the @AccessTimeout annotation is in. For a given class, the metadata for the @AccessTimeout annotation is never inherited from a class higher in the class inheritance tree.

  Like the @Lock annotation, you can also specify the @AccessTimeout annotation using the XML deployment descriptor, and if you use the XML deployment descriptor, the metadata from the @AccessTimeout annotation is ignored. The following example uses the XML deployment descriptor to specify the same metadata as the previous example.

```
<session>
  <ejb-name>ConfigurationBean</ejb-name>
  <concurrent-method>
    <method>
      <method-name>get</method-name>
    </method>
    <lock>Read</lock>
    <access-timeout>
      <timeout>1000</timeout>
      <unit>Milliseconds</unit>
    </access-timeout>
  </concurrent-method>
</session>
```

- It is important to know that the XML coding of the concurrent-methodType follows the three styles outlined in the EJB specification for composing the XML for container-transaction method elements.

  Remember that both the lock and access-timeout elements are optional. The three styles are described as follows:

  Style 1 uses the special method name * to apply the lock type, access-timeout value, or both to all business methods for the specified bean.

```
<!-- Example: Style 1 -->

  <concurrent-method>
    <method>
      <method-name>*</method-name>
    </method>
    <lock>Read</lock>
    <access-timeout>
      <timeout>2000</timeout>
      <unit>Milliseconds</unit>
    </access-timeout>
  </concurrent-method>
```

  Style 2 is used to refer to a business method with a specific name and assign it the specified lock type, access-timeout value, or both. If the method name is overloaded, meaning multiple methods have the

same name but different method signatures, all methods with this name have the specified lock type, access-timeout value, or both. Style 2 takes precedence over style 1.

```
<!-- Example: Style 2 -->

      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
        </method>
        <lock>Read</lock>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
```

Style 3 is used to refer to a distinct method that matches the given method name and has a method signature which matches the method parameters listed. Style 3 takes precedence over both style 1 and style 2.

```
<!-- Example: Style 3 -->

      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
          <method-params>
            <method-param>long</method-param>
            <method-param>int</method-param>
          </method-params>
        </method>
        <lock>Read</lock>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
```

If style 1 XML is used to define a lock type, then all @Lock annotations on the bean are ignored. The same is true for access timeout. If style 1 XML is used to define an access timeout value then all @AccessTimeout annotations on the bean are ignored.

- It is important to understand that the @Lock and @AccessTimeout annotations are treated independently of one another as is the corresponding XML deployment descriptor code for each.

  The implementation of this concept has multiple benefits. This separation of lock type and access timeout helps to prevent you from negatively affecting black box application code by not requiring you to know the lock type. You can safely leave the lock type value and adjust the access timeout value only to fit your environment needs, and avoid possible deadlock situations or other concurrency issues. Consider a scenario where you have a vendor supplied EJB application running and due to system slow down it keeps timing out. You do not want to change the locking logic for fear of causing a deadlock situation but you do want to modify the timeout limit. You can edit the deployment descriptor and specify the access timeout value you need for the methods you want to modify.

  The following example shows how you can specify just a method level @Lock(READ) annotation in the bean implementation and use style 2 for composing XML to specify the access-timeout element to be 2,000 milliseconds, not providing the optional lock element. The result is a method with a read lock that has an access timeout of 2,000 milliseconds.

```
@Singleton
public class ConfigurationBean implements Configuration {

 @Lock(READ)
 public Object businessMethod(long value) {
  // ...
 }

       // ...
}
```

```
    <session>
      <ejb-name>ConfigurationBean</ejb-name>
      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
        </method>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
    </session>
```

Similarly, you can use a class level lock annotation and then specify the wanted access timeout value in XML using either style 2, style 3, or both as shown in the following example.

```
@Singleton
@Lock(READ)
public class ConfigurationBean implements Configuration {

 public Object businessMethod(long value) {
  // ...
 }

 public Object businessMethod(long value, int i, Object value) {
  // ...
 }

 public Object businessMethod(long value, int i) {
  // ...
 }

}
```

```
    <session>
      <ejb-name>ConfigurationBean</ejb-name>
      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
        </method>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
          <method-params>
            <method-param>long</method-param>
            <method-param>int</method-param>
          </method-params>
        </method>
        <access-timeout>
          <timeout>8000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
    </session>
```

The previous code example results in all methods named “businessMethod” having a lock type of read and an access timeout of 2,000 milliseconds. The exception is the one instance of the method “businessMethod” that has a method signature with the first parameter being of type long and the second being of type int, This instance of method "businessMethod" has a lock type of read, but has an access timeout of 8,000 milliseconds. The same principle applies when style 1 XML is used to define

only a lock type, but not an access timeout value. You can add an access timeout value to a specific method or methods using style 2, style 3, or both to get a specific lock type and access timeout value result. The following example illustrates this point:

```
    <session>
      <ejb-name>ConfigurationBean</ejb-name>
      <concurrent-method>
        <method>
          <method-name>*</method-name>
        </method>
        <lock>Read</lock>
      </concurrent-method>
      <concurrent-method>
        <method>
          <method-name>businessMethod</method-name>
        </method>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
    </session>
```

The previous code example results in all business methods having a lock type of read and the method named, businessMethod, having a lock type of read and an access timeout of 2,000 milliseconds. You can also have a class level @Lock annotation to set the lock type for all methods and use style 1 for composing XML to set just the access timeout value for all methods. See the following example:

```
@Singleton
@Lock(READ)
public class ConfigurationBean implements Configuration {

 public Object businessMethod(long value) {
  // ...
 }

        // ...

}
```

```
    <session>
      <ejb-name>ConfigurationBean</ejb-name>
      <concurrent-method>
        <method>
          <method-name>*</method-name>
        </method>
        <access-timeout>
          <timeout>2000</timeout>
          <unit>Milliseconds</unit>
        </access-timeout>
      </concurrent-method>
    </session>
```

The previous example results in all business methods of bean, ConfigurationBean, having a lock type of read and an access timeout value of 2,000 milliseconds.

- Ensure that you understand the inheritance rules for the annotations that you have used.
- Avoid reentrant locking behavior that can occur if a read lock method calls a write lock method in the same singleton session bean.

  Suppose the business method of a singleton session bean either directly or indirectly causes another business method of the singleton session bean to be started. If the first method is a write lock method, then that method can call any other business method of the singleton session bean without any special consideration. However, you might carefully implement a read lock method if it were to call another business method of that same singleton session bean class that is a write lock method. Then a `javax.ejb.IllegalLoopbackException` exception occurs.

Changing singleton session bean locking policy

Use this task to override the default non-fair locking policy for all singleton session bean write locks within the server. This task is for WebSphere Application Server users who do not want lock requests for their Singleton session bean method invocations to follow a non-fair policy.

About this task

Locks for singleton session bean methods are obtained using a *non-fair* locking policy by default. When locks are constructed as *fair*, threads contend for entry using an approximate arrival-order policy. When lock is released, the longest waiting lock is granted regardless if it is reader or writer.

When locks are constructed as non-fair, the order in which locks are obtained is not guaranteed. If readers are active and a writer enters the queue, subsequent readers might get granted the read lock before the writer is granted the lock.

Procedure

1. Optional: Open the administrative console.
2. Select **Servers**.
3. Select **Server Types**.
4. Select **WebSphere application servers**.
5. Select the server that you want to configure.
6. From Server Infrastructure, select **Java and Process Management Process definition.**
7. From Additional Properties, select **Java Virtual Machine**.
8. In the Additional Properties area, select **Custom Properties**.
9. On the Application servers page, click **New** to specify an arbitrary name and value pair for your server.
10. In the **Name** entry field, type: `com.ibm.websphere.ejbcontainer.useFairSingletonLockingPolicy`
11. In the **Value** entry field, type `true`.

    Attention: Entering `true` causes all locks obtained for singleton session bean methods to use a fair policy. Entering `false` causes all locks obtained for singleton session bean methods to use a non-fair policy. The default policy is non-fair.
12. Click **OK**.
13. Save the configuration.
14. Restart the server.

Results

The locking policy has now been set for all locks obtained for singleton session beans within the server.

Programming to use message-driven beans

Applications can use message-driven beans as asynchronous message consumers. You deploy a message-driven bean as a message listener for a destination. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination.

About this task

You can use a tool such as Rational Application Developer to develop applications that use message-driven beans. You can use the WebSphere Application Server runtime tools, such as the administrative console, to deploy and administer applications that use message-driven beans.

For more information about implementing enterprise applications that use message-drive beans, see the following topics:

Procedure

- Develop message-driven beans.
- Design an enterprise application to use message-driven beans.
- Develop an enterprise application to use message-driven beans.
- Deploy an enterprise application to use message-driven beans with JCA 1.5-compliant resources.
- Deploy an enterprise application to use message-driven beans with listener ports.

Developing message-driven beans

You can develop a bean implementation class for a message-driven bean as introduced by the Enterprise JavaBeans specification. A message-driven bean (MDB) is a message consumer that implements business logic and runs on the server.

Before you begin

Determine the messaging model you want for your application regarding use of topics, queues, producers and consumers, publish or subscribe, and so on. You can refer to the message-driven bean component contract that is described in the Enterprise JavaBeans™ specification.

About this task

A message-driven bean (MDB) is a consumer of messages from a Java Message Service (JMS) provider. An MDB is invoked on arrival of a message at the destination or endpoint that the MDB services. MDB instances are anonymous, and therefore, all instances are equivalent when not actively servicing a client message. The container controls the life cycle of bean instances, which hold no state that is visible to a client.

The following example is a basic message-driven bean:

```
@MessageDriven(activationConfig={
                @ActivationConfigProperty(propertyName="destination",     propertyValue="myDestination"),
                @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
public class MsgBean implements javax.jms.MessageListener {

  public void onMessage(javax.jms.Message msg) {

      String receivedMsg = ((TextMessage) msg).getText();
      System.out.println("Received message: " + receivedMsg);

   }

}
```

As with other enterprise bean types, you can also declare metadata for message-driven beans in the deployment descriptor rather than using annotations, for example:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar id="EJBJar_1060639024453" version="3.0"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
      metadata-complete="false">
  <enterprise-beans>

    <message-driven>

      <ejb-name>MsgBean</ejb-name>
```

```
      <ejb-class>com.acme.ejb.MsgBean</ejb-class>
      <activation-config>
         <activation-config-property>
            <activation-config-property-name>destination</activation-config-property-name>
            <activation-config-property-value>myDestination</activation-config-property-value>
         </activation-config-property>
         <activation-config-property>
           <activation-config-property-name>destinationType</activation-config-property-name>
           <activation-config-property-value>javax.jms.Queue</activation-config-property-value>
        </activation-config-property>
      </activation-config>

    </message-driven>

  </enterprise-beans>
</ejb-jar>
```

Procedure

- Code the business logic of the message-driven bean, which must implement the appropriate message listener interface defined by the messaging type; for example, javax.jms.MessageListener. The business logic is invoked when the message listener method of the MDB is called to service a message; for example, `MessageListener.onMessage()`. If the MDB implements more than one interface, denote the message listener interface by coding the `messageListenerInterface` attribute of the `MessageDriven` annotation, or by coding the `<messaging-type>` element of the `message-driven` deployment descriptor element. You do not have to specify which is the message listener interface, as long as there is only one interface other than `java.io.Serializable`, `java.io.Externalizable`, or any of the `javax.ejb` package interfaces.
- You can optionally define message destination references on any type of enterprise bean. A message destination reference is a logical name by which an enterprise bean can refer to a message destination. The `Resource` annotation is used to inject a message destination reference, for example:

  ```
  @Resource (name="jms/Outlet", type=javax.jms.Queue) Queue salesOutlet;
  ```

  Alternatively, you can use the `<message-destination-ref>` element in the deployment descriptor to specify the message destination reference; for example:

  ```
  <message-destination-ref>
     <message-destination-ref-name>jms/Outlet</message-destination-ref-name>
     <message-destination-type>javax.jms.Queue</message-destination-type>
     <injection-target>
       <injection-target-class>com.acme.ejb.MsgBean</injection-target-class>
       <injection-target-name>salesOutlet</injection-target-name>
    </injection-target>
  </message-destination-ref>
  ```

  The `message-destination-ref` element is similar to the `resource-env-ref` element, but also has subelements, `message-destination-usage` with possible values `Produces`, `Consumes` or `ProducesConsumes`, and `message-destination-link`. You can use the `message-destination-link` element to tie two or more message-destination-ref references in the deployment descriptor together, which allows the deployer to bind the destination for several enterprise beans all at once, to the same destination. The message-destination-link value must match the `message-destination-name` value in the `message-destination` element; for example:

  ```
  <ejb-jar>

    <enterprise-beans>

      <session>
        <ejb-name>OutletBean</display-name>
        ...
        <message-destination-ref>
          <message-destination-ref-name>jms/target</message-destination-ref-name>
          <message-destination-type>javax.jms.Queue</message-destination-type>
          <message-destination-usage>Produces</message-destination-usage>
          <message-destination-link>destination</message-destination-link>
        </message-destination-ref>
        ...
  ```

```
    </session>

    <session>
      <ejb-name>InletBean</display-name>
      ...
      <message-destination-ref>
        <message-destination-ref-name>jms/source</message-destination-ref-name>
        <message-destination-type>javax.jms.Queue</message-destination-type>
        <message-destination-usage>Consumes</message-destination-usage>
        <message-destination-link>destination</message-destination-link>
      </message-destination-ref>
      ...
    </session>


    <message-driven>
      <ejb-name>InletBean</display-name>
      ...

      <ejb-name>MsgBean</ejb-name>
      <ejb-class>com.acme.MsgBean</ejb-class>
      <messaging-type>javax.jms.MessageListener</messaging-type>
      <message-destination-type>javax.jms.Queue</message-destination-type>
      <message-destination-link>destination</message-destination-link>

      ...
    </message-driven>

  </enterprise-beans>
      ...
  <assembly-descriptor>
    ...
    <message-destination>
      <message-destination-name>destination</message-destination-name>
    </message-destination>
    ...
  </assembly-descriptor>

</ejb-jar>
```

The `message-destination-link` element can refer to a destination that is defined in a different Java archive (JAR) file within the same application, as with an `ejb-link` element. For example, to link to the destination, `ProduceQueue`, defined in the `grocery.jar` file, enter the following line in the deployment descriptor:

```
<message-destination-link>grocery.jar#ProduceQueue</message-destination-link>
```

- As with any enterprise bean, you can package a message-driven bean in a JAR file, or in a web application archive (WAR) file.

Results

You developed a simple message-driven bean, along with some deployment and packaging options.

What to do next

Read related information about designing an enterprise application that uses message-driven beans.

Designing an enterprise application to use message-driven beans

To help you design your enterprise application, consider a generic enterprise application that uses one message-driven bean to retrieve messages from a JMS queue destination, and passes the messages on to another enterprise bean that implements the business logic.

About this task

To design an enterprise application to use message-driven beans, complete the following steps:

Procedure

1. Identify the message listener interface for the message type that the message-driven bean is to handle. The message-driven bean class must implement this message listener interface. For example, an EJB message-driven bean class used for JMS messaging must implement the javax.jms.MessageListener interface.
2. Identify the resources that the application is to use. This helps to identify the properties of resources that must be used within the application and configured as application deployment descriptors or within WebSphere Application Server.

Table 60. JMS resource types and examples of their properties. The left hand column of this table lists the JMS resource types, and the right hand column shows examples of the properties of each of the JMS resource types shown in the left hand column.

JMS resource type	Properties (for example)
JMS connection factory	**Name:** SamplePtoPQueueConnectionFactory **JNDI Name:** Sample/JMS/QCF
JMS destination	**Name:** Q1 **JNDI Name:** Sample/JMS/Q1
J2C activation specification properties	**Name:** MyMDBsActivationSpec **JNDI Name:** eis/MyMDBsActivationSpec **Destination JNDI Name:** MyQueue **Destination type:** javax.jms.Queue
Message-driven bean (deployment properties)	**Name:** JMSppSampleMDBBean **Transaction type:** Container **Message selector:** JMSType='car' **Acknowledge mode:** Dups OK Acknowledge **Destination type:** javax.jms.Queue **ActivationSpec JNDI name:** MyMDBsActivationSpec
Business logic bean	**Name:** MyLogicBean

   Ensure that you use consistent values where needed; for example, the JNDI name for the J2C activation specification must be the same in both the activation specification and the Message-driven bean deployment attributes.
3. Separate out the business logic. You should develop a message-driven bean to delegate the business processing of incoming messages to another enterprise bean. This provides clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client.
4. Decide whether to configure security. Messages arriving at a destination being processed by a listener have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component. For more information about EJB security, see EJB component security.
5. Understand how best effort nonpersistent messages are handled by the default messaging provider.

   If you have a non-transactional message-driven bean, the system either deletes the message when the message-driven bean starts, or when the message-driven bean completes. If the message-driven bean generates an exception, and therefore does not complete, the system takes one of the following actions:

   - If the system is configured to delete the message when the message-driven bean completes, then the message is despatched to a new instance of the message-driven bean, so the message has another opportunity to be processed.
   - If the system is configured to delete the message when the message-driven bean starts, then the message is lost.

   The message is deleted when the message-driven bean starts if the quality of service is set to `Best effort nonpersistent`. For all other qualities of service, the message is deleted when the message-driven bean completes.

Developing an enterprise application to use message-driven beans

Applications can use message-driven beans as asynchronous message consumers. You deploy a message-driven bean as a message listener for a destination. The message-driven bean is invoked by an activation specification or a JMS listener when a message arrives on the input destination that is being monitored.

About this task

You develop an enterprise application to use a message-driven bean as with any other enterprise bean, except that a message-driven bean does not have a home interface or a remote interface.

You should develop your message-driven bean to delegate the business processing of incoming messages to another enterprise bean, which provides clear separation of message handling and business processing. This separation also means that the business processing can be invoked either by the arrival of incoming messages or, for example, by a WebSphere J2EE client. Responses can be handled by another enterprise bean acting as a sender bean, or they can be handled in the message-driven bean.

EJB 2.0 message-driven beans support only Java Message Service (JMS) messaging. EJB 2.1 and EJB 3 message-driven beans can support other messaging types in addition to JMS. All message-driven beans must implement the MessageDrivenBean interface. For JMS messaging, a message-driven bean must also implement the message listener interface javax.jms.MessageListener. Other Java EE Connector Architecture (JCA)-compliant resource adapters might provide their own message listener interfaces that must be implemented.

You can use the New Enterprise Bean wizard of Rational Application Developer to create an enterprise bean with a bean type of `Message-driven bean`. The wizard creates appropriate methods for the type of bean.

By convention, the message-driven bean class is named *name*Bean, where *name* is the name you assign to the message-driven bean; for example:

```
public class MyJMSppMDBBean implements MessageDrivenBean, javax.jms.MessageListener
```

A message-driven bean can be registered with the EJB timer service for time-based event notifications if it also implements the javax.ejb.TimedObject interface, and invokes the timer callback method by the following call: `void ejbTimeout(Timer)`. At the scheduled time, the container then calls the message-driven bean ejbTimeout method.

The message-driven bean class must define and implement the following methods:

- onMessage(message), which must meet the following requirements:
  - The method must have a single argument of type javax.jms.Message.
  - The `throws` clause must not define any application exceptions.
  - If the message-driven bean is configured to use bean-managed transactions, it must call the javax.transaction.UserTransaction interface to scope the transactions. Because these calls occur inside the onMessage() method, the transaction scope does not include the initial message receipt. For more information, see the topic about message-driven beans transaction support.

  To handle the message within the onMessage() method (for example, to pass the message on to another enterprise bean), you use standard JMS. This is known as bean-managed messaging.

  If you are using a JCA-compliant resource adapter with a different message listener interface, another method besides the onMessage() method might be needed. For information about the message listener interface needed, see the documentation that was provided with your JCA-compliant resource adapter.
- ejbCreate()

  You must define and implement an ejbCreate method for each way in which you want a new instance of an enterprise bean to be created.
- ejbRemove()

This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean home interface from the javax.ejb.EJBHome interface. This method must contain any code that you want to execute before an enterprise bean instance is removed from the container (and the associated data is removed from the data source).

- ejbTimeout(Timer)

  This method is needed only to support notifications from the timer service, and contains the business logic that handles time events received.

Procedure

1. Create the Enterprise Application project.
2. Create the message-driven bean class. You can use the New Enterprise Bean wizard of Rational(r) Application Developer to create the enterprise bean with a bean type of `Message-driven bean`.

   For an example of how to create the message-driven bean class, see the Example section of this topic. For more information, see *Creating message-driven beans* in the Rational Application Developer information center. The result of this step is a message-driven bean that can be assembled into an EAR file for deployment.
3. Optional: Use the EJB deployment descriptor editor to review and, if needed, change the deployment attributes. You can use the EJB deployment descriptor editor to review deployment attributes that you specified on the EJB creation wizard (such as **`Transaction type`** and **`Message selector`**) and other default deployment attributes.

   If needed, you can override the values of these attributes later, after the enterprise application has been exported into an EAR file for deployment, as described in “Configuring deployment attributes for a message-driven bean against JCA 1.5-compliant resources” on page 2085 and “Configuring deployment attributes for a message-driven bean against a listener port” on page 2088.

   a. In the property pane, select the **Bean** tab.

   b. On the main panel, configure the **`Transaction type`** attribute.

      `Transaction type`
      This attribute determines whether the message-driven bean manages its own transactions, or whether the container manages transactions on behalf of the bean.
      `Bean` The message-driven bean manages its own transactions.
      `Container`
      The container manages transactions on behalf of the bean.

   c. Under **Activation Configuration**, review the following attributes:

      `acknowledgeMode`
      This attribute determines how the session acknowledges any messages it receives.
      `Auto Acknowledge`
      The session automatically acknowledges delivery of each message.
      `Dups OK Acknowledge`
      The session lazily acknowledges the delivery of messages. This setting is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumer applications that are tolerant of duplicate messages.

      As defined in the EJB specification, clients cannot use the Message.acknowledge() method to acknowledge messages. If a value of `CLIENT_ACKNOWLEDGE` is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and the Message.acknowledge() method is not used.

      Note:

      The acknowledgement is sent when the message is deleted.

      If you have a non-transactional message-driven bean, the system either deletes the message when the message-driven bean starts, or when the message-driven bean

completes. If the message-driven bean generates an exception, and therefore does not complete, the system takes one of the following actions:

- If the system is configured to delete the message when the message-driven bean completes, then the message is despatched to a new instance of the message-driven bean, so the message has another opportunity to be processed.
- If the system is configured to delete the message when the message-driven bean starts, then the message is lost.

The message is deleted when the message-driven bean starts if the quality of service is set to `Best effort nonpersistent`. For all other qualities of service, the message is deleted when the message-driven bean completes.

destinationType
This attribute determines whether the message-driven bean uses a queue or topic destination.
Queue The message-driven bean uses a queue destination.
Topic The message-driven bean uses a topic destination.

subscriptionDurability
This attribute determines whether a JMS topic subscription is durable or nondurable.
Durable
A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription messages until they are received by the subscription or until they expire.
Nondurable
Nondurable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A nondurable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created.

messageSelector
This attribute determines the JMS message selector that is used to select which messages the message-driven bean receives. For example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

d. Specify bindings deployment attributes.

Under **WebSphere Bindings**, select the **JCA Adapter** option then specify the bindings deployment attributes:
ActivationSpec JNDI name
This attribute specifies the JNDI name of the activation specification that is used to deploy this message-driven bean. This name must match the name of an activation specification that you define to WebSphere Application Server.
ActivationSpec Authorization Alias
This attribute specifies the name of an authentication alias used for authentication of connections to the JCA resource adapter. An authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.
Destination JNDI name
This attribute specifies the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI namespace.

e. Optional: Specify **Destination Link** to enable message linking.

Message linking allows the routing of messages to a specific message-driven bean in a deployment. Message linking allows message flow to be orchestrated between components in the same application.

For a message to be consumed and processed by a message-driven bean, the<message-destination-link> element must be defined in the deployment descriptor associated with the message-driven bean. The destination identified by the <message-destination-link> element corresponds to the logical destination.

When the <message-destination-ref> includes a <message-destination-link> element, messages are consumed at that destination.

In order to get the message-driven bean to consume messages sent to a destination, you can declare a <message-destination-link> element in the deployment descriptor, or alternatively set it in the activation specification.

4. Assemble and package the application for deployment.

Results

The result of this task is an EAR file, containing the message-driven bean, for the enterprise application that can be deployed in WebSphere Application Server.

Example

The following example shows how to create the message-driven bean class. The example code shows how to access the text and the JMS **MessageID**, from a JMS message of type TextMessage. In this example, first the onMessage() method of a message-driven bean is used to unpack the incoming text message and to extract the text and message identifier; then a private putMessage method (defined within the same message bean class) is used to put the message onto another queue:

```
public void onMessage(javax.jms.Message msg)
{
        String text      = null;
        String messageID = null;

        try
        {
                text = ((TextMessage)msg).getText();

                System.out.println("senderBean.onMessage(), msg text2: "+text);

                //
                // store the message id to use as the Correlator value
                //
                messageID = msg.getJMSMessageID();

                // Call a private method to put the message onto another queue
                putMessage(messageID, text);
        }
        catch  (Exception err)
        {
                err.printStackTrace();
        }
        return;
}
```

What to do next

After you have developed an enterprise application to use message-driven beans, configure and deploy the application. For example, define activation specifications for the message-driven beans and, optionally, change the deployment descriptor attributes for the application. For more information, see “Deploying an

enterprise application to use message-driven beans with JCA 1.5-compliant resources" on page 2084 and "Deploying an enterprise application to use message-driven beans with listener ports" on page 2088.

Assembling EJB 2.1 enterprise beans

Assembling EJB 2.x modules

An enterprise bean is a Java component that can be combined with other resources to create Java EE applications. This topic describes assembling Enterprise JavaBeans (EJB) modules based on the EJB 2.x and earlier specifications.

Before you begin

Create and unit tested an enterprise bean that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Assemble an EJB module to contain enterprise beans and related code artifacts. Group web components, client code, and resource adapter code in separate modules. After the EJB module is assembled, install it as a stand-alone application or combine it with other modules into an enterprise application.

Use an assembly tool to assemble an EJB module in any of the following ways:
- Import an existing EJB module (EJB JAR file)
- Create a EJB module
- Copy code artifacts, such as entity beans, from one EJB module into a new EJB module

For information about assembling EJB modules with an assembly tool, see the Rational Application Developer documentation.

Results

The EJB module is migrated or created, reflecting the Java EE structure that specifies the location of enterprise bean content files, class files, class paths, the deployment descriptor, and supporting metadata. For more information about the location of the content see the assembly tool information center.

What to do next

After you finish assembling your EJB module, you are ready to deploy your module.

Sequence grouping for container-managed persistence in assembled EJB modules

After assembling an Enterprise JavaBeans (EJB) module that contains container-managed persistence (CMP) beans, you can prevent certain types of database-related exceptions from occurring during application run time. Using *sequence grouping*, you can specify the order in which entity beans update relational database tables.

Note: Entity beans are not supported in EJB 3.x modules.

Eliminate exceptions resulting from referential integrity (RI) violations

Sequence grouping is particularly useful for preventing violations of database *referential integrity* (RI). A database RI policy prescribes rules for how data is written to and deleted from the database tables to

maintain relational consistency. Run-time requirements for managing bean persistence, however, can cause an EJB application to violate RI rules, which can cause database exceptions. These runtime requirements mandate that:

- Entity bean creates and remove operations correlate to the database immediately upon method invocation.
- Entity bean changes are cached by the EJB container until either a finder method is called, or the transaction ends.

Consequently, the order in which entity beans update the database is unpredictable. That randomness translates into high risk of the application violating database RI. Although caching the operations for batch processing overrides these runtime requirements, it does not guarantee a bean persistence sequence that follows any given RI policy.

The only way to guarantee a persistence sequence that honors database RI is to designate the sequence, which you do in the EJB deployment descriptor editor of the assembly tool. Through the sequence grouping feature, you assign beans to CMP groups. Within each group, you specify the order in which the persistence manager inserts bean data into the database to accomplish updates without violating RI.

See the “Setting the run time for CMP sequence groups” topic for detailed instructions on designating sequence groups. Consult your database administrator about the RI policy with which you need to synchronize.

Minimize exception risk for optimistic concurrency control schemes

Sequence grouping can also reduce the risk of transaction rollback exceptions for entity beans that are configured for optimistic concurrency control. In these concurrency control schemes, database locks are held for minimal amounts of time so that a maximum number of transactions consistently have access to the data. The relatively unrestricted state of the database can lead to transaction rollback exceptions for two common reasons:

- When concurrent transactions attempt to lock the same table row, database deadlock occurs.
- Transactions can occur in an order that violates application logic.

Use the sequence grouping feature to order bean persistence so that these scenarios are less likely to occur.

Setting the run time for CMP sequence groups

By designating container managed persistence (CMP) sequence groups for entity beans, you can prevent certain types of database-related exceptions during the run time of your Enterprise JavaBeans (EJB) application. Specify in each group the order in which the beans update your relational database tables.

Before you begin

When you define a sequence group, you designate it as one of two types:

- RI_INSERT, for setting a bean persistence sequence to prevent database referential integrity (RI) violations. See the topic Sequence grouping for container-managed persistence in assembled EJB modules for more information.
- UPDATE_LOCK, for setting a bean persistence sequence to minimize exceptions resulting from optimistic concurrency control

About this task

Both types of sequence groups must be created after you have assembled the beans into an EJB module, before installing your application on the product. If you need to edit sequence groups, uninstall the application, make your changes with an assembly tool, and reinstall your application.

Attention: If you already selected or plan to use top-down mapping for mapping your enterprise beans to back-end data, you do not need to create a sequence group with an RI_INSERT type. The product does not generate an RI policy for the database schema that it creates when you select top-down mapping.

To learn how to complete this task see the assembly tool information center.

What to do next

You are now ready to deploy your EJB module or combine it with other modules into a Java EE application. For more information about these two tasks, see the topics Installing enterprise application files and Assembling Java EE client applications.

Assembling EJB 3.x enterprise beans

EJB 3.0 and EJB 3.1 application bindings overview

Prior to starting an application that is installed on an application server, all Enterprise JavaBeans (EJB) references and resource references defined in the application must be bound to the actual artifacts (enterprise beans or resources) defined in the application server.

Starting in Version 8.0, bindings support in the EJB container is expanded. The EJB container assigns default JNDI bindings for EJB 3.x business interfaces based on application name, module name, and component name. You do not have to explicitly define JNDI binding names for each of the interfaces or EJB homes within an EJB 3.x module or no-interface views within an EJB 3.1 module.

When defining bindings, you specify Java Naming and Directory Interface (JNDI) names for the referenceable and referenced artifacts in an application. The jndiName values specified for artifacts must be qualified lookup names.

You do not need to manually assign JNDI bindings names for each of the interfaces, EJB homes, or no-interface views on your enterprise beans in EJB 3.x modules. If you do not explicitly assign bindings, the EJB container assigns default bindings.

Namespaces for default EJB JNDI bindings

Default EJB bindings may be placed by the application server into both the classic and java:[scope] sets of namespaces.

The set of classic namespaces consists of the ejblocal: and Global JNDI namespaces. The classic namespaces include a WebSphere extension, and they existed in the application server prior to version 8.0.

The set of java:[scope] namespaces consists of the java:global, java:app, and java:module namespaces. The java:[scope] namespaces are defined by the Java EE 6 specification, and are introduced into theWebSphere Application Server in Version 8. They are not a replacement for the classic namespaces. Rather, they are added in addition to the classic namespaces.

Classic namespace details

For the EJB 3.x level, the product provides two distinct classic namespaces for EJB interfaces, depending on whether the interface is local or remote. The same provision applies to EJB homes and no-interface views, which can be considered special types of interfaces. The following classic namespaces exist:

- Java virtual machine (JVM)-scoped, ejblocal: namespace
- Global JNDI namespace

Local EJB interfaces, homes, and no-interface views must be bound into a JVM-scoped classic ejblocal: namespace; they are accessible only from within the same application server process.

In contrast, remote EJB interfaces and homes must always be bound into the classic globally scoped WebSphere JNDI namespace; they can be accessed from anywhere, including other server processes and other remote clients. Local interfaces and no-interface views cannot be bound into the classic globally scoped JNDI namespace, nor can remote interfaces be bound into the JVM-scoped classic ejblocal: namespace.

The classic ejblocal: and classic globally scoped JNDI namespaces are separate and distinct. For example, an EJB local interface or no-interface view bound at “ejblocal:AccountHome” is a not the same as a remote interface bound at “AccountHome” in the classic globally scoped namespace. This behavior helps maintain the distinction between your local and remote interface references. Having a JVM-scoped local namespace also makes it possible for your applications to directly look up or reference local EJB interfaces and no-interface views from anywhere in the JVM server process, including across Java Platform, Enterprise Edition (Java EE) application boundaries.

Default classic JNDI bindings for EJB business interfaces in the EJB 3.x container with the Application, Module, and Component names

The EJB container assigns default classic JNDI bindings for EJB 3.x business interfaces based on application name, module name, and component name, so it is important to understand how these names are defined. Each of these names is a character string.

Java EE applications are packaged in a standardized format called an Enterprise Application Archive (EAR) file. The EAR is a packed file format like a .zip or .tar file format, and can thus be visualized as a collection of logical directories and files packed together into a single physical file. Within each EAR file are one or more Java EE module files, which can include:

- Java Application Archive (JAR) files for EJB modules, Java EE application client modules and utility class modules
- Web Application Archive (WAR) files for web modules
- Other technology-specific modules such as Resource Application Archive (RAR) files and other types of modules

Within each module file are typically one or more Java EE components. Examples of Java EE components are enterprise beans, servlets, and application client main classes.

Since Java EE modules are packaged within Java EE application archives, and Java EE components are in turn packaged within Java EE modules, the “nesting path” of each component can be used to uniquely identify every component within a Java EE application archive, according to its application name, module name, and component name.

Application name used in classic bindings

The name of an application is defined by the following (in order of priority):

- The value of the application name specified to the product administrative console, or the **appname** parameter supplied to the **wsadmin** command-line scripting tool, during installation of the application into the product.
- The value of the **<display-name>** parameter within the `META-INF/application.xml` deployment descriptor for the application.
- The EAR file name, excluding the `.ear` file suffix. For example, an application EAR file named `CustomerServiceApp.ear` would have an application name of “CustomerServiceApp” in this case.

The Java EE 6 specification provides for EJB JNDI lookup names of the general form, java:global[/*appName*]/*moduleName*/*beanName*. The *appName* component of the lookup name is shown

as optional because it does not apply to beans deployed in stand-alone modules. Only beans packaged in .ear files contain the *appName* component in the java:global lookup name. The rules that determine the value for *appName* are different from the rules described previously for application names. The *appName* value in the java:global lookup name template shown previously is defined by the following, in order of priority:

- The value of the **<application-name>** parameter within the `application.xml` deployment descriptor for the application.
- The EAR file name, excluding the `.ear` file suffix. For example, an application EAR file named `CustomerServiceApp.ear` has an application name of "CustomerServiceApp". If the application is a stand-alone module, the java:global lookup name does not contain an application component.

The value for *appName* is also the string value bound under the name, java:app/AppName, in accordance with the Java EE 6 specification.

Module name used in classic bindings

The name of a module is defined as the Uniform Resource Identifier (URI) of the module file, relative to the root of the EAR file in which it resides. Stated another way, the module name is the file name of the module relative to the root of the EAR file, including any subdirectories in which the module file is nested. This naming convention is still true even when a logical module name is explicitly specified using the module-name element in the deployment descriptor.

In the following example, the "CustomerServiceApp" application contains three modules whose names are `AccountProcessing.jar`, `Utility/FinanceUtils.jar`, and `AppPresentation.war`:

```
CustomerServiceApp.ear:AccountProcessing.jar
com/mycompany/AccountProcessingServiceBean.class AccountProcessingService.class
Utility/FinanceUtils.jar META-INF/ejb-jar.xml
com/mycompany/InterestCalculatorServiceBean.class InterestCalculatorService.class
AppPresentation.war META-INF/web.xml
```

The Java EE 6 specification provides for EJB JNDI lookup names of the general form, java:global[/*appName*]/*moduleName*/*beanName*. The *appName* component of the lookup name is shown as optional because it does not apply to beans deployed in stand-alone modules. Only beans packaged in `.ear` files contain the *appName* component in the java:global lookup name. Another JNDI lookup name variant that includes the module name is java:app/*moduleName*/*beanName*. The value for *moduleName* is not the module URI. The *moduleName* value in the java:global and java:app lookup name templates is defined by the following, in order of priority:

- The value of the **<module-name>** parameter within the `ejb-jar.xml` or `web.xml` deployment descriptor for the module.
- The module URI, excluding its `.jar` or `.war` suffix. For example, a module with a URI of `CustomerService.jar` or `CustomerService.war` has a module name of "CustomerService".

The value for *moduleName* is also the string value bound under the name, java:module/*moduleName*, in accordance with the Java EE 6 specification. This also applies to client modules. For client modules, the **<module-name>** parameter is located in the `application-client.xml` deployment descriptor file.

EJB component name used in classic bindings

The name of an EJB component is defined by the following value, in order of priority:

- The value of the ejb-name tag associated with the bean in the `ejb-jar.xml` deployment descriptor, if present.
- The value of the "name" parameter, if present, in the @Stateless or @Stateful annotation associated with the bean.
- The name of the bean implementation class, without any package-level qualifier.

Bindings

Review the following bindings that are supported by EJB 3.x modules:
- Default classic bindings for business interfaces, homes, and no-interface views
- Default classic binding pattern
- java:[scope] bindings
- User-defined bindings for EJB business interfaces, homes, and no-interface views
- User-defined bindings for resolving references and injection targets
- Default resolution of EJB references and EJB injections: The AutoLink feature
- Resolution of EJB and resource references and injections: The lookup feature
- “Overriding environment entries defined in applications ” on page 451
- “Overriding data source definitions ” on page 451
- Naming considerations in clustered environments
- User-defined EJB extension settings
- Legacy (XMI) bindings
- User-specified XML bindings

Default classic bindings for EJB business interfaces, homes, and no-interface views

You do not have to explicitly define JNDI binding names for each of the interfaces or EJB homes within an EJB 3.x module or no-interface views within an EJB 3.1 module. If you do not explicitly assign bindings, the EJB container of the product assigns default classic bindings using the rules outlined here. This is different from the EJB support in the product prior to the EJB 3.0 specification being supported.

The EJB container performs two default classic bindings for each interface (business, remote home, or local home) or no-interface view on each enterprise bean. These two classic bindings are referred to here as the views of the interface or no-interface *short* binding and *long* binding. The short binding uses just the package-qualified Java class name of the interface or no-interface view, while the long binding uses the component ID of the enterprise bean as an extra qualifier prior to the package-qualified interface or no-interface view class name, with a hash or number sign (# symbol) between the component ID and the interface or no-interface view class name. You can think of the difference between the two forms as being analogous to a *short* TCP/IP host name (just the machine name) versus a *long* host name (machine name with domain name prepended to it).

For example, the short and long default classic bindings for an interface or no-interface view might be com.mycompany.AccountService and AccountApp/module1.jar/ServiceBean#com.mycompany.AccountService, respectively.

By default, the component ID for EJB default classic bindings is formed using the application name, module name, and component name of the enterprise bean, but you can assign any string you want instead. By defining your own string as the component ID, you can set up a naming convention where the long-form bindings of the enterprise bean share a common user-defined portion, yet also have a system-defined portion based on the name of each interface or no-interface view class. It also allows you to make the default EJB binding names independent of how you have packaged the enterprise beans within the application module hierarchy. Overriding a default component ID of an enterprise bean is described in the “User-defined bindings for EJB business interfaces, homes, and no-interface views” section of this topic.

As mentioned earlier in the section on the classic JVM-scoped local namespace and the classic globally scoped JNDI namespace, all local interfaces, homes, and no-interface views must be bound into the classic ejblocal: namespace, which is accessible only within the same server process (JVM), while remote interfaces and homes must be bound into the classic globally scoped namespace, which is accessible from anywhere in the WebSphere product cell. As you would expect, the EJB container follows these rules for the default bindings.

In addition, the *long* default bindings for remote interfaces follow recommended Java EE best practices in that they are grouped under an ejb context name. By default, EJB remote home and business interfaces are bound into the root of the application server naming context. However, the application server root context is used for binding more than just EJB interfaces, so to keep this context from getting too cluttered, it is a good practice to group EJB-related bindings into a common “ejb” subcontext rather than placing them directly in the server root context. It is like why you would use subdirectories on a disk volume rather than putting all the files in the root directory.

The *short* default bindings for remote interfaces are not bound in the ejb context. The short default bindings are located in the root of the server root context. Even though it is a best practice to group all the EJB-related bindings under an ejb context, there are other considerations including the following situations:

- The short default bindings provide a simple, direct way to access an EJB interface. Placing them directly in the server root context and referring to them by just the interface name or the interface name prepended with ejblocal: was in keeping with that goal of simplicity.
- At the same time, placing the long default bindings in the ejb context, or the ejblocal: context in the case of a local interface, kept those bindings out of the root context of the server and reduced the clutter there enough to allow having the short bindings in the root context.
- It provides a degree of cross-compatibility with other Java EE application servers that use similar naming conventions.

To summarize, all local default bindings, both short and long, are placed in the classic ejblocal: server/JVM-scoped namespace, while remote default bindings are placed in the root context of the server of the classic globally scoped namespace if they are short, or in the <server_root>/ejb context (following the root context of the server) if they are long. Thus, the only default bindings in the globally scoped root context of the server are the short bindings for remote interfaces, which is the best balance between providing a simple, portable usage model and keeping the globally scoped root context of the server from becoming too cluttered.

Default classic binding pattern

The patterns for each type of classic binding are displayed in the table. In these patterns, strings written in *<bracketed italics>* represent a value. For example, *<package.qualified.interface>* might be com.mycompany.AccountService, and *<component-id>* might be AccountApp/module1.jar/ServiceBean.

Table 61. Default binding patterns. Default binding patterns

Binding patterns	Description
ejblocal:*<package.qualified.interface>*	Short form local interfaces, homes, and no-interface views
<package.qualified.interface>	Short form remote interfaces and homes
ejblocal:*<component-id>*#*<package.qualified.interface>*	Long form local interfaces, homes, and no-interface views
ejb/*<component-id>*#*<package.qualified.interface>*	Long form remote interfaces and homes

The *component-id* default pattern is *<application-name>*/*<module-jar-name>*/*<ejb-name>* unless it is overridden in the EJB module binding file using the component-id attribute as described in the next section, “Conflicts in short default binding names when multiple enterprise beans implement the same interface”. The *<module-jar-name>* variable, when not overridden by the EJB module binding file, is the name of the physical module file within the EAR including the extension, for example, `.jar`, `.ear`, `.war`, as described in the previous “Module name” section, even if a logical module name is specified in the deployment descriptor.

Conflicts in short default classic binding names when multiple enterprise beans implement the same interface

When more than one enterprise bean that is running in the application server implements a given interface or no-interface view, the short default classic binding name becomes ambiguous because the short name might refer to any of the Enterprise JavaBeans that implement this interface or no-interface view. To avoid

this situation, you must either explicitly define a binding for each Enterprise JavaBeans that implements the given interface or no-interface view as described in the next section, or disable short default classic bindings for applications containing these Enterprise JavaBeans by defining a WebSphere product JVM custom property, com.ibm.websphere.ejbcontainer.disableShortDefaultBindings. For more information about defining the JVM custom property, read about Java Virtual machine custom properties.

To use this JVM custom property, set the property name to com.ibm.websphere.ejbcontainer.disableShortFormBinding and the property value to either * (asterisk) as a wildcard value to disable short form default classic bindings for all applications in the server, or to a colon-delimited sequence of the Java EE application names for which you want to disable short default classic bindings, for example, PayablesApp:InventoryApp:AccountServicesApp.

Effect of explicit assignment on default classic bindings

If you explicitly assign a binding definition for an interface, home, or no-interface view, no short or long default classic bindings are performed for that interface or no-interface view.

Note: This only applies to the specific interfaces or no-interface views for which you assign an explicit binding. Other interfaces on that enterprise bean, without explicitly assigned bindings, are bound by using default classic binding names.

java:[scope] namespaces

The java:global, java:app, and java:module namespaces are introduced by the Java EE 6 specification. They provide a mechanism for binding and looking up resources that are portable across application servers.

The server always creates a default long-form binding for each EJB interface, including the no-interface view, and places them into the java:global, java:app, and java:module namespaces. A short-form binding is also created and placed into the java:global, java:app, and java:module namespaces, if the bean exposes only one interface, including the no-interface view. The default bindings are only created for session beans. They are not created for entity beans or message driven beans.

The long-form and short-form bindings both contain the application name, the module name, and the bean component name. The application name is defaulted to the base name of the .ear file, without the extension. The application name can be overridden using the application-name element in the application.xml file. The module name is defaulted to the path name of the module, with the extension removed and any directory names included. The module name can be overridden using the module-name element in the ejb-jar.xml or web.xml files. The bean component name defaults to the unqualified name of the bean class. The bean component name can be overridden using the name attribute on the EJB component defining annotation, or the ejb-name element in the ejb-jar.xml file.

The long-form binding pattern is java:global/*<applicationName>*/*<moduleName>*/*<bean component name>*!*<fully qualified interface name>*.

The short-form binding pattern is java:global/*<applicationName>*/*<moduleName>*/*<bean component name>*.

For example, the bean component MyBeanComponent exposes just the one com.foo.MyBeanComponentLocalInterface interface, and is packaged in the `myModule.jar` module in the `myApp.ear` file. As a result, the following bindings are created in the java:[scope] namespaces:

- java:global/myApp/myModule/MyBeanComponent!com.foo.MyBeanComponentLocalInterface
- java:global/myApp/myModule/MyBeanComponent
- java:app/myModule/MyBeanComponent!com.foo.MyBeanComponentLocalInterface
- java:app/myModule/MyBeanComponent

- java:module/MyBeanComponent!com.foo.MyBeanComponentLocalInterface
- java:module/MyBeanComponent

The MyBeanComponent bean can be obtained from the java:[scope] namespaces using one of the following techniques:

- Use the lookup attribute on the @EJB annotation; for example:
  ```
  @EJB(lookup="java:global/myApp/myModule/MyBeanComponent")
  ```
- Use the lookup-name element in ejb-jar.xml; for example:
  ```
  <lookup-name>java:global/myApp/myModule/MyBeanComponent!com.ibm.MyBeanComponentLocalInterfaces</lookup-name>
  ```
- Complete a lookup on the InitialContext object; for example:
  ```
  initialContext.lookup("java:global/myApp/myModule/MyBeanComponent!com.foo.MyBeanComponentLocalInterfaces")
  ```

In addition to the default bindings created by the application server, you can define references in the java:global, java:app, and java:module namespaces. References defined in the java:global, java:app, and java:module namespaces do not go into the component namespace. References defined in the java:global, java:app, or java:module namespaces must be looked up or injected from those namespaces. They cannot be looked up or injected from the component namespace.

A bean component can use the java:module namespace to declare a reference that is usable by a component packaged in the same module. It can use the java:app namespace to declare a reference that is usable by a component packaged in a different module within the same application. It can use the java:global namespace to declare a reference that is usable by a component packaged in a different application.

References with identical names in the java:global, java:app, or java:module namespaces might conflict with each other, just as references with identical names in the component namespace might conflict. A reference scoped to the java:app namespace for one application does not conflict with an identically named reference scoped to the java:app namespace for a different application. Likewise, a reference scoped to the java:module namespace for one module does not conflict with an identically named reference scoped to the java:module namespace for a different module.

A reference can be declared in the java:global namespace using annotations; for example:

```
@EJB(name="java:global/env/myBean")
```

A reference can be declared in the ejb-jar.xml file; for example:

```
<resource-ref>
    <res-ref-name>java:global/env/myDataSource</res-ref-name>
    ....
</resource-ref>
```

For additional documentation on the java:[scope] namespaces, see section 5.2.2 of the Java EE 6 specification and section 4.4 of the Enterprise JavaBeans 3.1 specification.

User-defined bindings for EJB business interfaces, homes, and no-interface views

For cases where you want to manually assign binding locations rather than using the product default bindings, you can use the EJB module binding file to assign your own binding locations to specific interfaces, homes, and no-interface views. You can also use this file to only override the component ID portion of the default bindings on one or more enterprise beans in the module. Overriding the component ID provides a middle ground between using all default bindings and completely specifying the binding name for each interface or no-interface view.

To specify user-defined bindings information for EJB 3.x modules, place the file ibm-ejb-jar-bnd.xml, in the META-INF directory for the EJB Java archive (JAR) file. The suffix on this file is XML. Also, when defining

a classic binding for a local interface or no-interface view, you must preface the name with the string “ejblocal:” so it is bound into the classic JVM-scoped ejblocal: namespace.

The `ibm-ejb-jar-bnd.xml` file is used for EJB 3.0 and later modules that run on the product, whereas the `ibm-ejb-jar.bnd.xmi` file is used for pre-EJB 3.0 modules and for web modules. The binding file format in the `ibm-ejb-jar.bnd.xml` file is different from the XMI file format for the following reasons:

- Bindings and extensions that are declared in the XMI file format depend on the presence of a corresponding `ejb-jar.xml` deployment descriptor file that explicitly refers to unique ID numbers that are attached to elements in that file. This system is no longer viable for EJB 3.0 and later modules, where it is no longer a requirement for the module to contain an `ejb-jar.xml` deployment descriptor.
- The XMI file format was designed to be machine-edited only by the product development tools and system management functions; it was effectively part of the internal implementation of the product and the file structure was never documented externally. This made it impossible for developers to manually edit binding files, or create them as part of a WebSphere independent build process, in a supported way.
- Rather than referring to encoded ID numbers in the `ejb-jar.xml` deployment descriptor, the XML-based binding file refers to an EJB component by its EJB name. Each EJB component in a module has a unique EJB name, either by default or through explicit assignment by the developer. Therefore, this behavior provides an unambiguous way to target bindings and extensions.
- The new binding files are XML-based, and an XML Schema Definition (XSD) file is provided to externally document the structure. These `.xsd` files can be consumed by many common XML file editors to assist in syntactic verification and code completion functions. As a result, it is now possible for developers to produce and edit the binding and extension files independently of the application server infrastructure.

The following table lists the `ibm-ejb-jar-bnd.xml` elements and attributes that are used to assign bindings to EJB interfaces and homes for EJB 3.x modules and no-interface views for EJB 3.1 modules.

Table 62. `ibm-ejb-jar-bnd.xml` elements and attributes. `ibm-ejb-jar-bnd.xml` elements and attributes

Element or attribute	How used	Example	Comments
<session>	Declares a group of binding assignments for a session bean.	`<session name="AccountServiceBean"/>`	Requires name attribute and at least one of the following attributes: simple-binding-name attribute, local-home-binding-name attribute, remote-home-binding-name attribute, or <interface> element.
name	Attribute that identifies the ejb-name of the enterprise bean that a <session>, <message-driven>, or <entity>, or other element applies to.	`<session name="AccountServiceBean"/>`	The name value is the name declared in the <ejb-name> element of an ejb-jar.xml deployment descriptor file, the name attribute of a @Stateful, @Stateless, @Singleton, or @MessageDriven annotation, or defaults to the unqualified class name of the EJB implementation class annotated with the @Session or @MessageDriven annotation (if no <ejb-name> value is declared in the XML deployment descriptor and no name parameter is declared on the annotation).

Table 62. `ibm-ejb-jar-bnd.xml` elements and attributes (continued). `ibm-ejb-jar-bnd.xml` elements and attributes

Element or attribute	How used	Example	Comments
component-id	Attribute that overrides the default component ID value for an enterprise bean. The default long-form classic bindings for this enterprise bean uses the specified component ID instead of *<app_name>/<module_jar_name>/<bean_name>*.	`<session name="AccountServiceBean" component-id="Dept549/AccountProcessor"/>` The previous example results in the bean whose ejb-name is AccountServiceBean, having its long-form default classic local interfaces bound at ejblocal:Department549/AccountProcessor#*<package.qualified.interface>* Its long-form default classic remote interfaces are bound at ejb/Department549/AccountProcessor#*<package.qualified.interface>*	Can be used alone, or in combination with the <interface> element, the local-home-binding-name attribute, or the remote-home-binding-name attribute. Interfaces that are not assigned explicit bindings might have default classic bindings performed using the user-specified component ID value. Interfaces that are assigned explicit bindings are bound using those values. Since the simple-binding-name attribute is intended to apply to all defined interfaces on a given enterprise bean (leaving no interfaces defaulted), applying a component-id in combination with a simple-binding-name is typically not useful.

Table 62. `ibm-ejb-jar-bnd.xml` elements and attributes (continued). `ibm-ejb-jar-bnd.xml` elements and attributes

Element or attribute	How used	Example	Comments
simple-binding-name	A simple mechanism for assigning interface bindings for Enterprise JavaBeans that: • Implement a single EJB 3.x business interface • Implement a pre-EJB 3.0 style component interface (local, remote or both types) with a companion EJB home. The value of the attribute is used as the binding location of the enterprise bean business interface, or the binding location of the Enterprise JavaBeans local, remote homes, or both. The binding is placed in the classic ejblocal: namespace if the interface or home is local, and placed in the root context of the application server of the classic globally scoped JNDI namespace if the interface or home is remote.	`<session name="AccountServiceBean" simple-binding-name="ejb/AccountService"/>` This example results in the bean whose ejb-name is AccountServiceBean, having its local business interface or home, if any, bound at ejblocal:ejb/AccountService in the classic local JVM-scoped EJB namespace, and its remote business interface or home (if any) bound at ejb/AccountService in the root context of the application server of the classic globally scoped JNDI namespace. **Important:** Important:The exact value of the attribute, including, in this specific example, the “ejb” subcontext name is used even if the interface is a local interface bound into the ejblocal: namespace. When user-defined bindings are specified, the exact name specified by the attribute is used.)	Not to be used in combination with local-home-binding-name or remote-home-binding-name attributes, or the <interface> element. Also, must not be used on beans that implement more than one business interface - use the <interface> element in that case instead. If this attribute is used on an enterprise bean that implements more than one business interface, or a combination of business interface and local/remote component interface with home, the resulting bindings are disambiguated by appending a hash or number sign (# symbol) to the attribute value, followed by the package-qualified class name of each interface, home, or both on the enterprise bean. This condition can be avoided, however, by using the <interface> element to define a binding for each of the business interfaces instead of using simple-binding-name. **Important:** Important: defining a simple-binding-name on a bean that implements more than one business interface is not the same as overriding the default component ID for a bean using <component-id>. Remote interface default bindings defined with a component-id are still grouped under the EJB context (as all remote interface default bindings are), while remote interface bindings disambiguated by the EJB container in response to erroneous use of simple-binding-name on a bean with multiple interfaces are not grouped under the ejb context.Additionally, the inclusion of the package-qualified class name always occurs for long-form default classic bindings, whereas with simple-binding-name it occurs only on error conditions, where disambiguation is necessary. Do not depend on the binding name created through disambiguation, since whether that effect occurs might change if the bean is changed to implement more or fewer interfaces.
local-home-binding-name	Attribute to specify the binding location of the local home of an enterprise bean.	`<session name="AccountServiceBean" local-home-binding-name="ejblocal:AccountService"/>`	Not to be used in combination with the simple-binding-name attribute. Since local homes must always be bound into the classic JVM-scoped namespace, the value must begin with the ejblocal: prefix.

Table 62. ibm-ejb-jar-bnd.xml elements and attributes (continued). ibm-ejb-jar-bnd.xml elements and attributes

Element or attribute	How used	Example	Comments
remote-home-binding-name	Attribute to specify the binding location of the remote home of an enterprise bean.	<session name="AccountServiceBean" remote-home-binding-name= "ejb/services/AccountService"/>	Not to be used in combination with the simple-binding-name attribute. The value cannot begin with the classic ejblocal: prefix, since remote homes cannot be bound into the classic ejblocal: namespace.
<interface>	A subelement of the <session> element that assigns a binding to a specific EJB business interface or no-interface view. In contrast to the simple-binding-name, local-home-binding-name and remote-home-binding-name attributes, both a binding-name parameter and a class parameter are necessary (in fact, this distinction is why a separate XML element is necessary rather than an attribute). The class parameter specifies the package-qualified name of the business interface or no-interface view class to be bound.	<interface class="com.ejbs.InventoryService" binding-name="ejb/Inventory"/> (declared as a subelement inside a <session> element)	Not to be used in combination with the simple-binding-name attribute. Since local interfaces and no-interface views must always be bound into the classic JVM-scoped namespace, the binding-name value must begin with the ejblocal: prefix when this element is applied to a local interface or no-interface view.
binding-name	Attribute to specify the binding location of a business interface bound with the <interface> element.	<interface class="com.ejbs.InventoryService" binding-name="ejb/Inventory"/> (declared as a subelement inside a <session> element)	Required in combination with the <interface> element (and used on that element only). Since local interfaces must always be bound into the classic JVM-scoped namespace, the binding-name value must begin with the ejblocal: prefix when applied to a local interface.

Binding file Example 1

The following example is a basic ibm-ejb-jar-bnd.xml file that contains only the elements and attributes that assign binding names to EJB interfaces and no-interface views. It overrides the component ID used for default bindings on the enterprise bean that is named *S01*, and assigns explicit bindings to some of the interfaces on the enterprise beans, *S02* and *S03*, in this module.

```
<?xml version="1.0" encoding="UTF-8?">
<ejb-jar-bnd xmlns=http://websphere.ibm.com/xml/ns/javaee xmlns:xsi="
 http://www.w3.org/2001/XMLSchema-instance "xsi:schemaLocation"=
 http://websphere.ibm.com/xml/ns/javaee
 http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd "version 1.0">
 <session name="S01" component-id="Department549/AccountProcessors"/>
 <session name="S02" simple-binding-name="ejb/session/S02"/>
 <session name="S03">
  <interface class="com.ejbs.BankAccountService" binding-name="ejblocal:session/BAS"/>
 </session>
</ejb-jar-bnd>
```

The binding file has the following results:

- The session bean with ejb-name S01 is assigned a user-defined component ID, overriding the default component ID (application name/ejb-jar module name/bean name) for all interfaces and no-interface views on that bean. Local interfaces and no-interface views on this bean are bound at `ejblocal:Department549/AccountProcessors#<package.qualified.interface.name>` while remote interfaces are bound at `ejb/Department549/AccountProcessors#<package.qualified.interface.name>`
- The session bean with ejb-name *S02* is assumed to have a single EJB 3.x business interface or EJB 3.1 no-interface view. Alternatively, it could have a pre-EJB 3.0 “component” interface with local home, remote home, or both local and remote homes. The business interface, or the home or homes of the component interface are bound at ejblocal:ejb/session/S02 if it is local, or ejb/session/S02 if it is remote.

If bean *S02* has more than one business interface, or business interfaces and home, a simple-binding-name is ambiguous. In that case, the container disambiguates the binding assignments by appending #<*package.qualified.interface.name*> to the simple binding name, ejb/session/S02, for each of the bean interfaces.

- The EJB 3.x business interface or EJB 3.1 no-interface view, com.ejbs.BankAccountService, on the session bean with ejb-name *S03* is bound at ejblocal:session/BAS.

All other business interfaces, homes, and no-interface views on this bean, if present, are assigned default classic bindings. The com.ejbs.BankAccountService interface is assumed to be local since it was designated for the ejblocal: namespace in this example; an error occurs if the interface is not local.

The next section expands on this example, introducing elements for resolving the targets of various kinds of reference and injection entries that are declared either in the XML deployment descriptor or through annotations.

User-defined bindings for resolving references and injection targets

The previous section showed examples of assigning user-defined binding names for business interfaces, homes, and no-interface views. This section covers resolving linkage targets for references, injection directives, and message-driven bean destinations.

Table 63. Elements and attributes to resolve linkage targets for references and injection targets. Elements and attributes to resolve linkage targets for references and injection targets

Element or attribute	How used	Example	Comments
<jca-adapter>	Defines the JCA 1.5 adapter activation spec, and a message-destination JNDI location, for delivery of messages to a message-driven bean.	`<jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/>`	Requires *activation-spec-binding-name* attribute. If the corresponding message-drive bean does not identify its message destination by using the <message-destination-link> element, then the *destination-binding-name* attribute is also required. Can optionally include activation-spec-auth-alias attribute.
<ejb-ref>	Resolves the target of an ejb-ref declaration, which is declared through the @EJB annotation or through the ejb-ref in the ejb-jar.xml deployment descriptor, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target enterprise bean in the classic JVM-scoped ejblocal:, or classic globally scoped JNDI namespace.	`<ejb-ref name="com.ejbs.BankAccountServiceBean/s02Ref" binding-name="ejb/session/S02"/>`	Requires the name and binding-name attributes.
<message-driven>	Declares a group of binding assignments for a message-driven bean.	`<message-driven name="EventRecorderBean"> <jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/> </message-driven>`	Requires name attribute and <jca-adapter> subelement.

Table 63. Elements and attributes to resolve linkage targets for references and injection targets (continued).
Elements and attributes to resolve linkage targets for references and injection targets

Element or attribute	How used	Example	Comments
<message-destination>	Associates the name of a message destination, which is a logical name defined in a Java EE module deployment descriptor, with a specific global JNDI name, which is an actual name in the JNDI namespace. <message-destination-ref> elements in the Java EE module deployment descriptor, or @Resource injection directives that inject message destinations, can then use the <message-destination-line> element to refer to this message-destination by the destination logical name, rather than requiring individual <message-destination-ref> binding entries in the binding file for each defined message-destination-ref.	`<message-destination name="EventProcessingDestination" binding-name="jms/ServiceQueue"/>`	Requires name and binding-name attributes.
<message-destination-ref>	Resolves the target of a message-destination-ref declaration that is declared through the @Resource annotation or through the message-destination-ref in ejb-jar.xml, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target resource environment in the global JNDI namespace.	`<message-destination-ref name="com.ejbs.BankAccountServiceBean/serviceQueue" binding-name="jms/ServiceQueue"/>`	Requires the name and binding-name attributes.
<resource-ref>	Resolves the target of a resource-ref declaration that is declared through the @Resource annotation or through resource-ref in ejb-jar.xml, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target resource in the global JNDI namespace.	`<resource-ref name="com.ejbs.BankAccountServiceBean/dataSource" binding-name="jdbc/Default"/>`	Requires the name and binding-name attributes. Can include the authentication-alias or custom-login-configuration attributes.
<resource-env-ref>	Resolves the target of a resource-env-ref declaration that is declared through the @Resource annotation or through resource-env-ref in ejb-jar.xml, providing the linkage between the name declared in the component-scoped java:comp/env namespace and the name of the target resource environment in the global JNDI namespace.	`<resource-env-ref name="com.ejbs.BankAccountServiceBean/dataFactory" binding-name="jdbc/Default"/>`	Requires the name and binding-name attributes.
<env-entry>	Overrides an environment entry with the specified value represented in string format or object which can be accessed with a JNDI lookup on the specified lookup name applied to the default initial context.	`<env-entry name="java:module/env/taxYear" value="2010"/>` `<env-entry name="java:module/env/taxYear" binding-name="cell/persistent/MyApp/MyModule/taxYear"/`	Requires the name attribute and either the value or the binding-name attribute, but not both.

Table 63. Elements and attributes to resolve linkage targets for references and injection targets (continued).
Elements and attributes to resolve linkage targets for references and injection targets

Element or attribute	How used	Example	Comments
<data-source>	Overrides a data source definition, which is declared through the @DataSourceDefinition annotation or through the data-source element in the application, or a module deployment descriptor, with a managed resource.	`<data-source name="java:module/env/myDS" binding-name="jdbc/DB2DS"/>`	Requires the name and binding-name attributes.
name	Attribute that identifies the naming location, typically within the component-specific java:comp/env namespace, that defines the “source” side of a reference/target linkage, such as in ejb-ref, resource-ref, resource-env-ref, message-destination, or message-destination-ref.	`<ejb-ref name="com.ejbs.BankAccountServiceBean/goodBye" binding-name="ejb/session/S02"/>`	
binding-name	Attribute that identifies the naming location within the classic ejblocal: or classic globally scoped JNDI namespace, or java:global namespace that defines the “target” side of a reference/target linkage, such as in ejb-ref, resource-ref, resource-env-ref, message-destination, or message-destination-ref.	`<ejb-ref name="com.ejbs.BankAccountServiceBean/goodBye" binding-name="ejb/session/S02"/>`	
value	Attribute that specifies the value to use for an env-entry binding.	`<env-entry name="java:module/env/taxYear" value="2010"/>`	
activation-spec-binding-name	Attribute that identifies the JNDI location of the activation specification associated with the JCA 1.5 adapter to be used to deliver messages to a message-driven bean.	`<jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/>`	This name must match the name of a JCA 1.5 activation specification that you define to WebSphere Application Server.
activation-spec-auth-alias	Optional attribute that identifies the name of a J2C authentication alias used for authentication of connections to the JCA resource adapter. A J2C authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.	`<jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" activation-spec-auth-alias="jms/Service47Alias" destination-binding-name="jms/ServiceQueue"/>`	This name must match the name of a J2C authorization alias that you define to WebSphere Application Server
destination-binding-name	Attribute that identifies the JNDI name that the message-driven bean uses to look up its JMS destination in the JNDI name space.	`<jca-adapter activation-spec-binding-name="jms/InternalProviderSpec" destination-binding-name="jms/ServiceQueue"/>`	This name must match the name of a JMS queue or topic that you define to WebSphere Application Server.
authentication-alias	Optional subelement of the <resource-ref> binding element. If the resource reference is for a connection factory, then an optional JAAS login configuration can be specified; in this case a simple authentication alias name.	`<resource-ref name="com.ejbs.BankAccountServiceBean/dataSource" binding-name="jdbc/Default"> <authentication-alias name="defaultAuth"/> <resource-ref>`	This name must match the name of a JAAS authentication alias that you define to WebSphere Application Server.
custom-login-configuration	Optional subelement of the <resource-ref> binding element. If the resource reference is for a connection factory, then an optional JAAS login configuration can be specified; in this case a set of properties (name/value pairs).	`<resource-ref name="com.ejbs.BankAccountServiceBean/dataSource" binding-name="jdbc/Default"> <custom-login-configuration-name="customLogin"> <property name="loginParm1" value="ABC123"/> <property name="loginParm2" value="DEF456"/> </custom-login-configuration> </resource-ref>`	This name must match the name of a JAAS login configuration that you define to WebSphere Application Server.

Binding file Example 2

The following example is an expansion of the basic ibm-ejb-jar-bnd.xml file introduced in Example 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd xmlns="http://websphere.ibm.com/xml/ns/javaee" "xmlns:xsi"=
"http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee"
"http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd version" "1.0">
 <session name="S01" component-id="Department549/AccountProcessors"/>
 <session name="S02" simple-binding-name="ejb/session/S02"/>
 <session name="S03">
  <interface class="com.ejbs.BankAccountService"
   binding-name="ejblocal:session/BAS"/>
  <ejb-ref name="com.ejbs.BankAccountServiceBean/goodBye"
   binding-name="ejb/session/S02"/>
  <resource-ref name="com.ejbs.BankAccountServiceBean/dataSource"
   binding-name="jdbc/Default"/>
 </session>
 <message-driven name="MO1">
  <jca-adapter activation-spec-binding-name="jms/InternalProviderSpec"
   destination-binding-name=jms/"ServiceQueue"/>
 </message-driven>
 <session name="S04" simple-binding-name="ejb/session/S04">
  <resource-ref name="ejbs.S04Bean/dataSource"
   binding-name="jdbc/Default">
   <authentication-alias name="defaultlogin"/>
  </resource-ref>
 </session>
 <session name="S05">
  <interface class="com.ejbs.InventoryService"
   binding-name="ejb/session/S05Inventory"/>
  <resource-ref name="ejbs.S05Bean/dataSource"
   binding-name="jdbc/Default">
   <custom-login-configuration name="customLogin">
    <property name="loginParm1" value="ABC123"/>
    <property name="loginParm2" value="DEF456"/>
   </custom-login-configuration>
  </resource-ref>
 </session>
</ejb-jar-bnd>
```

This binding has the following results:

1. The business interface, home, and no-interface view bindings for the session beans named S01, S02, and S03 are unchanged from the previous example.
2. The session bean whose ejb-name is *S03* now includes two reference target resolution bindings:
   - The ejb-ref binding resolves the EJB reference defined at java:comp/env/com.ejbs.BankAccountServiceBean/goodBye, to the JNDI location ejb/session/S02 within the root JNDI context of the application server. The EJB reference can also be defined by an @EJB injection in the class com.ejbs.BankAccountServiceBean, into an instance variable named “goodBye”.

     Note: ejb/session/S02 is the JNDI location of session bean *S02* also defined in this same binding file, which means that the reference points to the session bean whose name is *S02*.
   - The resource-ref binding resolves the resource reference defined at java:comp/env/com.ejbs.BankAccountServiceBean/dataSource, to the JNDI location jdbc/Default. The resource reference could also have been defined by a @Resource injection in the class com.ejbs.BankAccountServiceBean, into an instance variable named “dataSource”.
3. Bindings are defined for a message-driven bean whose ejb-name is *M01*. The MDB receives messages from a JMS destination defined to WebSphere Application Server, whose JNDI name is jms/ServiceQueue, using a JCA 1.5 adapter whose JCA 1.5 activation spec has been defined to WebSphere Application Server with the name jms/InternalProviderSpec.
4. The session bean whose ejb-name is *S04* is assumed to have a single business interface or no-interface view, which is bound at ejb/session/S04, if remote or ejblocal:ejb/session/S04, if local. It has a resource-ref with name, java:comp/env/ejbs/S04Bean/dataSource. This can also be the class, ejbs.S04Bean, with an @Resource injection into a variable named, dataSource. This resource-ref resolved to the JNDI location jdbc/Default. The resource-ref refers to a J2C connection and connects to this resource using a simple authentication alias named *defaultlogin* that has been defined in WebSphere Application Server.

5. A business interface binding is defined for the interface whose class name is com.ejbs.InventoryService implemented by the session bean whose ejb-name is *S05*; the interface is assumed to be remote since it is not prefixed with “ejblocal:” and might thus be bound at ejb/session/S05Inventory in the root JNDI context of the server in the classic globally-scoped namespace. Any other business interfaces implemented by this bean are assigned default classic bindings. The bean has a resource-ref with name java:comp/env/ejbs.S05Bean/dataSource (or a @Resource injection in the class ejbs.S05Bean into a variable named “dataSource”) that is resolved to the JNDI location jdbc/Default. The resource-ref refers to a J2C connection and connects to this resource using a custom login configuration that includes two name-value pairs.

Bindings file Example 3

This example later in this section demonstrates how to define and resolve EJB reference bindings to perform JNDI lookups across application server instances within the same WebSphere Application Server cell. It uses two EJB beans: a called bean that defines an explicit binding using the simple-binding-name attribute, and a calling bean that performs an @EJB injection and uses the ejb-ref element within its associated binding file to resolve the reference so it points at the called bean, that resides in a different application server process.

ibm-ejb-jar-bnd.xml (called bean)

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd xmlns="http://websphere.ibm.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee"
 "http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd" version="1.0">
 <session name="FacadeBean" simple-binding-name="ejb/session/FacadeBean"/>
</ejb-jar-bnd>
```

This binding file content assumes that the session bean whose ejb-name is “FacadeBean” implements a single business interface, and thus the simple-binding-name attribute can be used as an alternative to the <interface> subelement. In this case, the FacadeBean implements a single remote business interface, bound at ejb/session/FacadeBean in the server root JNDI context of the application server where the FacadeBean resides.

Code snippet (calling bean)

```
@EJB(name="ejb/FacadeRemoteRef")
FacadeRemote remoteRef;
try {
     output = remoteRef.orderStatus(input);
}
catch (Exception e) {
     // Handle exception, etc.
}
```

This code snippet performs an EJB resource injection into the instance variable named “remoteRef”, which is of type FacadeRemote. The injection overrides the “name” parameter, setting the resulting ejb-ref reference name to ejb/FacadeRemoteRef. The code invokes a business method on the injected reference.

ibm-ejb-jar-bnd.xml (calling bean)

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd xmlns="http://websphere.ibm.com/xml/ns/javaee"
 "xmlns:xsi="
 "http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee"
 "http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd" version="1.0">
 <session name="CallingBean">
  <ejb-ref name="ejb/FacadeRemoteRef"
   binding-name="cell/nodes/S35NLA1/servers/S35serverA1/ejb/session/FacadeBean"/>
 </session>
</ejb-bnd-jar>
```

Finally, this binding file resolves the EJB reference with an ejb-ref name of ejb/FacadeRemoteRef to point to the classic globally scoped JNDI name of cell/nodes/S35NLA1/servers/S35serverA1/ejb/session/FacadeBean. This classic globally scoped JNDI name represents an interface bound at ejb/session/FacadeBean under the server root context of the server named “S35serverA1” on the node

named “S35NLA1” within the WebSphere Application Server cell of the calling bean. To point to a location within a different WebSphere Application Server cell, a CORBAName-style name can be used instead of a standard JNDI name.

Instructions on how to modify the ibm-ejb-jar-bnd.xml file can be found in the topic, Ways to update application files.

The relationship between injections and references

There is a one-to-one correspondence between injection directives and reference declarations - every injection implicitly defines a reference of some type, and conversely, every reference can optionally also define an injection. You can think of an injection annotation as being the mechanism to define references through annotations rather than defining them in the XML deployment descriptor.

By default, an injection defines a reference with a name formed from the package-qualified class name of the component performing the injection, a forward slash (/), then the name of the variable or property being injected into. For example, an injection performed in the class com.ejbs.AccountService, into a variable or property named “depositService”, results in a reference named java:comp/env/com.ejbs.AccountService/depositService. However, specifying the optional “name” parameter on the injection directive overrides this default name and causes the reference to be named according to the value of the “name” parameter.

Knowing this rule, it is easy to see how a bindings file can be used not only to resolve targets for references declared in an XML deployment descriptor, but also to resolve targets for references implicitly declared by an annotation injection directive. Simply use the value of the “name” parameter on the injection annotation, or the default reference name from the class name and variable/property name if no “name” parameter is specified, as if it were the name of the reference declared in an XML deployment descriptor.

Default resolution of EJB references and EJB injections: The EJBLink and AutoLink features

The *EJBLink* and *AutoLink* features are two different mechanisms that resolve references to EJB components that are packaged in the same application and application server process as the referring component. Both EJBLink and AutoLink remove the need to explicitly resolve the EJB reference with binding information. The EJBLink feature is defined by the EJB specification, while the AutoLink feature is a WebSphere Application Server extension.

The EJBLink and AutoLink features use different search criteria to locate the targeted bean component. EJBLink searches for the targeted bean component using the explicitly specified bean name. AutoLink searches for the targeted bean component using the interface that the bean implements. If no explicit bindings are provided, but a bean name is provided, then the EJBLink feature is used. If no explicit bindings are provided, and no bean name is provided, then the AutoLink feature is used. The EJBLink and AutoLink features are never used together as part of the same search process.

Except for the search criteria, the EJBLink and AutoLink features are similar. Both features search a specific module first, and then if needed fall back to searching the other modules in the same application and application server process. Both features require that the search criteria resolve to exactly one bean component, and consider it an error condition when the search criteria resolves to multiple bean components. An error condition exists because the application server does not know that which of the multiple bean components must be used. In this case, the exception com.ibm.websphere.ejbcontainer.AmbiguousEJBReferenceException occurs. This exception is thrown at run time when the referencing component attempts to find the targeted bean component.

The EJBLink feature supports three different formats.

- Specify only the name of the bean component. For example, MyBean.

- Specify the name of the physical module file, including the extension, that contains the targeted bean component, followed by the # character, followed by the name of the bean component. For example, myModule.jar#MyBean
- Specify the logical name of the module that contains the targeted bean component, the slash character (/), followed by the name of the bean component. For example, MyModule/MyBean.

You can optionally specify the logical name of the module using the module-name element in the EJB deployment descriptor for an ejb-jar module, or you can use the module-name element in the web deployment descriptor file for a WAR module that contains EJB content. For a WAR module that contains EJB content, the module-name element specified in the EJB deployment descriptor is ignored, and the module-name element specified in the web deployment descriptor is processed. When no module-name value is specified in the deployment descriptor, a default logical name is assigned to the module. The default logical module name is the base name of the module file, minus the extension. For example, the module file, MyModule.jar, has the default logical module name MyModule.

Specifying the name of the physical module file is still supported even when the module has a logical name. Specifying the logical name of the module is still supported even when no logical module name is configured in the deployment descriptor. In this case, the base name of the module is used as the logical name of the module.

The embeddable EJB container supports all EJBLink formats. To support the physical module file format, the embeddable EJB container does not allow you to start multiple modules with the same base name.

AutoLink is a value-add feature of WebSphere Application Server that eliminates the need to explicitly resolve EJB reference targets in certain usage scenarios. In WebSphere Application Server V8, AutoLink is implemented within the boundaries of each WebSphere Application Server process. The AutoLink algorithm works as follows.

When the EJB container in the product encounters an EJB reference within a given EJB module, it first checks to see if you have explicitly resolved the target of that reference through inclusion of an entry in the binding file of the module. If it finds no explicit resolution of the target in the binding file, the container searches within the referring module for an enterprise bean that implements the interface type or no-interface view you have defined within the reference.

If it finds *exactly one* enterprise bean within the module that implements the interface or no-interface view, it uses that enterprise bean as the target for the EJB reference. If the container cannot locate an enterprise bean of that type within the module, it expands the search scope to the application that the module is part of, and searches other modules within that application that are assigned to the same application server as the referring module. Again, if the container finds *exactly one* enterprise bean that implements the target interface or no-interface view, within the other modules of the application assigned to the same server as the referring module, it uses that enterprise bean as the reference target.

The scope of AutoLink is limited to the application in which the EJB reference appears, and to the application server on which the referring module is assigned. References to enterprise beans in a different application, enterprise beans in a module assigned to a different application server, or to enterprise beans residing in a module that has been assigned to a WebSphere Application Server cluster, must be explicitly resolved using reference target bindings in the `ibm-ejb-jar-bnd.xml` file of the EJB module, or the `ibm-web-bnd.xmi` file of the web module.

It is important to note that AutoLink is only supported for EJB references, not other types of references although it is supported from the EJB container, the web container, and the application client container. Also, because the scope of the AutoLink function is limited to the server that the referring module is assigned to, or in the case of the Java EE client container, to the server that the client container is configured as its JNDI bootstrap server, it is useful mainly in development environments and other single-server usage scenarios. Even with these present limitations, it can be a significant value during the development experience by removing the need to explicitly resolve EJB references.

Resolution of EJB and resource references and injections: The lookup feature

The lookup feature is defined by the EJB 3.1 specification as a mechanism that resolves references to EJBs or resources, by an explicit JNDI name. You can specify the lookup attribute on the javax.ejb.EJB annotation or on the javax.annotation.Resource annotation. The corresponding XML attribute in the `ejb-jar.xml` file is <lookup-name>, on one of the following elements: <ejb-ref>, <ejb-local-ref>, <env-entry>, <resource-ref>, <resource-env-ref>, or <message-destination-ref>. lookup or <lookup-name> is a JNDI name relative to the java:comp/env naming context.

On an EJB reference, lookup or <lookup-name> must not be specified with beanName or with <ejb-link>. The administrative console displays lookup-name and ejb-link as read-only. However, if a JNDI name is specified in the application installation step “Map EJB references to beans”, it overrides the lookup-name or ejb-link value.

Overriding environment entries defined in applications

Applications might define environment entries with values that are suitable for unit testing, but not for integration testing or production use. If you want to override an environment entry value, you can add the following element to the corresponding binding file:

```
<env-entry name="name" value="value"/>
```

where *name* is the env-entry name as it is defined in the application and *value* is the value assigned to the env-entry represented in string format. The string for *value* is parsed according to the type of the environment entry as if the value had been specified in the deployment descriptor using env-entry-value. For example,

```
<env-entry name="java:module/env/taxYear" value="2010"/>
```

associates the env-entry named “java:module/env/taxYear” with a value of “2010”.

Alternatively, you can configure an env-entry as a reference to another object accessible through JNDI. The object returned from the JNDI lookup is used as the env-entry value. The element for that option has the following form:

```
<env-entry name="name" binding-name="lookupName"/>
```

where *name* is the env-entry name as it is defined in the application and *lookupName* is a JNDI name that resolves when applied to a lookup on the default initial context. For example,

```
<env-entry name="java:module/env/taxYear" binding-name="cell/persistent/MyApp/MyModule/taxYear"/>
```

associates the env-entry named “java:module/env/taxYear” with a value returned from a default initial context lookup operation on “cell/persistent/MyApp/MyModule/taxYear”. You are responsible for creating the JNDI object binding. The class of the bound object must be assignable to the object type of the associated env-entry.

Environment entries can be defined at the application level and in EJB, web, and client modules. Those levels correspond to the binding files `application-bnd.xml`, `ejb-jar-bnd.xml`, `web-app-bnd.xml`, and `application-client-bnd.xml`.

Overriding data source definitions

With Java EE 6, you can develop applications that define data sources using the @DataSourceDefinition annotation or <data-source> deployment descriptor entry.

Your applications should look up resource references as opposed to looking up data source definitions directly. If you are installing an existing application that contains direct lookups to a data source definition,

and you want it to use another data source definition, you can override the data source definition with a binding that resolves to a managed resource that you create. Create the binding by adding the following element to your binding file:

```
<data-source name="name" binding-name="lookupName"/>
```

where *name* is the env-entry name as it is defined in the application and *lookupName* is a JNDI name that resolves when applied to a lookup on the default initial context. For example,

```
<data-source name="java:module/env/myDS" binding-name="jdbc/DB2DS"/>
```

causes lookups on “java:module/env/myDS” to resolve to the data source bound with the name, “jdbc/DB2DS”, relative to the default initial context. The data source bound under jdbc/DB2DS can be created, for example, through the administrative console.

Data sources can be defined at the application level and in EJB, web, and client modules. Those levels correspond to the binding files `application-bnd.xml`, `ejb-jar-bnd.xml`, `web-app-bnd.xml`, and `application-client-bnd.xml`.

Naming considerations in clustered and cross-server environments

The classic global JNDI naming conventions in the previous sections apply in non-clustered environments and when the lookup target is within the same cluster as the source of the lookup. When a lookup is performed from outside a cluster on a binding that is within a given cluster, the lookup string must be qualified to indicate the name of the cluster in which the target resides, according to the following convention:

```
cell/clusters/<cluster-name>/<name-binding-location>
```

For example, given an EJB interface binding location within the application server root context:

```
ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

If the EJB implementing this interface is assigned to an application server that is a member of a cluster named Cluster47, the lookup string external to that cluster is as follows:

```
cell/clusters/Cluster47/ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

When a lookup is performed across application server processes, the lookup string must be qualified to indicate the name of the node and server in which the target resides, according to the following convention:

```
cell/nodes/<node-name>/servers/<server-name>/<name binding location>
```

Again, given an EJB interface binding location within the application server root context:

```
ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

If the enterprise bean that is implementing this interface is assigned to an application server named Server47A1 that is located on a node named S47NLA1, the cross-server lookup string is as follows:

```
cell/nodes/S47NLA1/servers/Server47A1/ejb/Department549/AccountProcessors/CheckingAccountReconciler
```

User-defined EJB extension settings

For cases where you want to specify values for WebSphere Application Server EJB Extension settings, you can use an *EJB module extension file* to assign these settings to specific EJB types within that module. You specify extension settings information for EJB 3.x modules by placing one, or both, of two files into the META-INF directory for the EJB JAR file, depending on the type of extension being defined. The names of the two files are `ibm-ejb-jar-ext.xml` and `ibm-ejb-jar-ext-pme.xml`.

Note: The suffix on these files are XML, not XMI as in prior versions of WebSphere Application Server.

The ibm-ejb-jar-ext.xml and ibm-ejb-jar-ext-pme.xml files are used for EJB 3.x modules running in WebSphere Application Server, whereas the ibm-ejb-jar-ext.xmi and ibm-ejb-jar-ext-pme.xmi files are used for pre-3.0 EJB modules. WebSphere Application Server Version 8.0 uses a new XML-based extension file format instead of the previous .xmi file format for the following reasons:

1. Bindings and extensions declared in the xmi file format depend on the presence of a corresponding ejb-jar.xml deployment descriptor file, explicitly referring to unique ID numbers attached to elements in that file. This system is no longer viable for EJB 3.0 and later modules, where it is no longer a requirement for the module to contain an ejb-jar.xml deployment descriptor.
2. The xmi file format was designed to be machine-edited only by WebSphere development tools and system management functions; it was effectively part of WebSphere's internal implementation and the structure of the file was never documented externally. This made it impossible for developers to manually create or edit binding or extension files, or create them as part of a WebSphere independent build process, in a supported manner.
3. Rather than referring to encoded ID numbers in ejb-jar.xml, the XML-based extension file format refers to EJB components by their EJB name. Each EJB component in a module is guaranteed to have a unique EJB name, either by default or through explicit assignment by the developer, so this provides an unambiguous way to target bindings and extensions.
4. The new binding and extension file formats are XML-based, and XML Schema Definition (xsd) files are provided to externally document their structure. These .xsd files might be consumed by many common XML file editors to assist in syntactic verification and code completion functions. As a result, it is now possible for developers to produce and edit these binding and extension files independently of WebSphere Application Server infrastructure, using a generic XML editor or scripting system of their choice.

Extensions defined in META-INF/ibm-ejb-jar-ext.xml

The following tables outlines extension elements and attributes that must be placed in the META-INF/ibm-ejb-jar-ext.xml file. The subsequent section lists elements and attributes that appear in a separate file, META-INF/ibm-ejb-jar-ext-pme.xml.

Table 64. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes
<session>	Declares a group of extension settings for a session bean.	<session name="AccountServiceBean"/>	Requires *name* attribute. In order to have any effect, also include at least one extension setting definition subelement.
<message-driven>	Declares a group of extension settings for a message-driven bean.	<message-driven name="EventProcessorBean"/>	Requires *name* attribute. In order to have any effect, also include at least one extension setting definition subelement.

Table 65. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes
<time-out>	Subelement to the <session> element that optionally declares the number of seconds between method invocations after which a *stateful* session bean might no longer be available.	<session name="ShoppingCartBean"> <time-out value="600"/> </session>	Requires *value* attribute, a positive integer. **Note:** Only applicable to stateful session beans; must not be used on stateless beans. Attribute default: 300 (5 minutes)

Table 65. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file (continued). Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes
<bean-cache>	Subelement of <session> element used to declare bean activation/passivation settings for stateful session beans.	<session name="ShoppingCartBean"> <bean-cache activation-policy="TRANSACTION"/> </session>	To have any effect, also include the activation-policy attribute.
activation-policy	Attribute of <bean-cache> element that declares the conditions under which the bean instance might be activated and passivated. Applicable to stateful session beans. Allowable values and their meanings are: • *TRANSACTION:* Indicates that the bean activates at the start of a transaction and passivates (and is removed from the active EJB instance cache) at the end of the transaction. • *ONCE:* Indicates that the bean activates when it is first accessed in the server process, and passivates (and is removed from the active EJB instance cache) at the discretion of the container, for example, when the cache becomes full. • *ACTIVITY_SESSION:* Indicates that the bean activates and passivates as follows: 1. On an ActivitySession boundary, if an ActivitySession context is present on activation 2. On a transaction boundary, if a transaction context (but no ActivitySession context) is present on activation, or 3. on an invocation boundary.	<session name="ShoppingCartBean"> <bean-cache activation-policy="ONCE"/> </session>	Attribute default: ONCE for stateful session beans.

Table 66. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes
<global-transaction>	Subelement to the <session> and <message-driven> elements that can be used to declare the transaction timeout (in seconds) to be used on transactions started by this specific EJB type (overriding the server setting for global transaction timeout) and also might declare whether this EJB type propagates global transaction context received through web service atomic transactions, across the heterogeneous web service environment.	<session name="AccountServiceBean" <global-transaction transaction-timeout="180" send-wsat-context="FALSE"/> </session>	Requires at least one of transaction-timeout or send-wsat-context attributes. Attribute default: Server transaction timeout setting for transaction-timeout; FALSE for send-wsat-context

Table 66. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file (continued). Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes

<local-transaction>	Subelement to the <session> and <message-driven> elements that can be used to declare settings related to local transactions. Supported attributes are boundary, resolver, and unresolved-action; these attributes configure, for the component, the behavior of the local transaction containment (LTC) environment of the container that the container establishes whenever a global transaction is not present. The meaning of each attribute is as follows:
	
	Boundary
	
	This setting specifies the containment boundary at which all contained resource manager local transactions (RMLTs) must be completed. Possible values are:
	• *BEAN_METHOD:* This is the default value. If you select this option, RMLTs must be resolved within the same bean method in which they were started.
	• *ACTIVITY_SESSION:* RMLTs must be resolved within the scope of any ActivitySession in which they were started or, if no ActivitySession context is present, within the same bean method in which they were started.
	
	Resolver
	
	This setting specifies the component responsible for initiating and ending RMLTs. Possible values are:
	• *APPLICATION:* This is the default value. The application is responsible for starting RMLTs and for completing them within the local transaction containment (LTC) boundary. Any RMLTs that are not completed by the end of the LTC boundary are cleaned up by the container according to the value of the Unresolved action attribute.
	• *CONTAINER_AT_BOUNDARY:* The container is responsible for starting RMLTs and for completing them within the LTC boundary. The container begins an RMLT when a connection is first used within the LTC scope, and completes it automatically at the end of the LTC scope. If Boundary is set to ActivitySession, the RMLTs are enlisted as ActivitySession resources and directed to complete by the ActivitySession. If Boundary is set to BeanMethod, the RMLTs are committed at the end of the method by the container.
	
	Unresolved Action
	
	This setting specifies the direction that the container requests RMLTs to take, if those transactions are unresolved at the end of the LTC boundary scope and the Resolver is set to Application. Possible values are:
	• *ROLLBACK:* This is the default value. At end of the LTC boundary scope, the container instructs all unresolved RMLTs to roll back.
	• *COMMIT:* At the end of the LTC boundary scope, the container instructs all unresolved RMLTs to commit. The container instructs the RMLTs to commit only in the absence of an unhandled exception. If the application method that is running in the local transaction context ends with an exception, any unresolved RMLTs are rolled back by the container. This is the same behavior as for global transactions.
		`<session`
		` name>="AccountServiceBean">`
		` <local-transaction`
		`  boundary="BEAN_METHOD"`
		`  resolver="APPLICATION"`
		`  unresolved-action="ROLLBACK"/>`
		`</session>`
			Requires at least one of boundary, resolver, or unresolved-action attributes.
			
			Attribute default:
			`boundary="BEAN_METHOD";`
			`resolver="APPLICATION";`
			`unresolved-action=`
			`"ROLLBACK"`

Table 67. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes

<method>	Sub-element to the <method-session-attribute> and <run-as-mode> elements that is used to specify the method name, method signature, or method types to which a given setting might apply. Supported attributes are type, name, and params. Each attribute is described as follows:
	
	type
	• *UNSPECIFIED:* The setting applies to all methods matching the name and params attributes, regardless of interface type.
	• *REMOTE:* The setting applies to remote business interface and remote component interface methods matching the name and params attributes.
	• *LOCAL:* The setting applies to local business interface, local component interface methods, and no-interface views that match the name attribute, params attribute, or both.
	• *HOME:* The setting applies to remote home interface methods matching the name and params attributes matching the name and params attributes.
	• *LOCAL_HOME:* The setting applies to local home interface methods matching the name and params attributes.
	• *SERVICE_ENDPOINT*: The setting applies to methods on the JAX-RPC service endpoint interface matching the name and params attributes.
	
	name
	
	The name of the method to which the setting is applied, or an asterisk (*) if the setting is to be applied to all methods regardless of name.
	
	params
	
	The parameter signature of the method to which the setting is applied. This can be used to uniquely qualify a particular method in cases where more than a single method uses the same name. The parameter signature is a comma-separated list of Java types. Primitive types are specified using their name only; non-primitive types are specified using their fully qualified class or interface name including any Java package, and arrays of Java types are specified by the type of the array element followed by one or more pair of square brackets (for example int[][]).
		`<session`
		` /name="AccountServiceBean">`
		` <method-session-attribute`
		`  type="REQUIRES_NEW">`
		` <method`
		`  type="LOCAL"`
		`  name="debitAccount"`
		`  params="java.lang.String[], int,`
		`  com.xyz.CustomerInfo"/>`
		` </method-session-attribute;>`
		`</session>`	

<run-as-mode>	Sub-element to the <session> and <message-driven> elements that can be used to declare the security identity that a given EJB method might have while the method is being executed. The identity can be set to use the identity of the caller (mode = CALLER_IDENTITY), the identity of the EJB server (mode = SYSTEM_IDENTITY), or the identity of a specific security role (mode = SPECIFIED_IDENTITY).	`<session`
		` name="AccountServiceBean">`
		` <run-as-mode`
		`  mode="SPECIFIED_IDENTITY">`
		` <specified-identity`
		`  role="admin"/>`
		` <method type="UNSPECIFIED"`
		`  name="testRunAsRole"/>`
		` </run-as-mode>`
		`</session>`
			Requires mode attribute and <method> subelement. If the mode is SPECIFIED_IDENTITY, the <specified-identity subelement is also required.

<start-at-app-start>	Subelement to the <session> and <message-driven> elements that can be used to inform the EJB container that specified EJB type might be initialized at the time the application is first started, rather than the time the EJB type is first used by the application.	`<session`
		` name="AccountServiceBean">`
		` <start-at-app-startvalue="TRUE"/>`
		`</session>`
			Requires value attribute.
			
			Attribute default: FALSE (initialize EJB type when EJB is first used by application) for beans other than message-driven beans. Always TRUE for message-driven beans.

Table 67. Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file (continued). Elements and attributes of the META-INF/ibm-ejb-jar-ext.xml file

Element or Attribute	Description	Example	Usage notes
<resource-ref>	Subelement to the <session> and <message-driven> elements, that might be used to declare additional settings on a Java EE resource reference, such as isolation level to be used on transactions driven through the connection referred to by the reference. Allowable attributes include *isolation-level*. The attributes are defined as follows: *isolation-level* • *TRANSACTION_REPEATABLE_READ:* This isolation level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads. • *TRANSACTION_READ_COMMITTED:* This isolation level prohibits dirty reads, but allows nonrepeatable reads and phantom reads. • *TRANSACTION_READ_UNCOMMITTED:* This isolation level allows reading uncommitted changes (data changed by a different transaction that is still in progress). It also allows dirty reads, nonrepeatable reads, and phantom reads. • *TRANSACTION_SERIALIZABLE:* This isolation level prohibits the following types of reads: 1. Dirty reads, in which a transaction reads a database row containing uncommitted changes from a second transaction, 2. Nonrepeatable reads, in which one transaction reads a row, a second transaction changes the same row, and the first transaction rereads the row and gets a different value, and 3. Phantom reads, in which one transaction reads all rows that satisfy an SQL WHERE condition, a second transaction inserts a row that also satisfies the WHERE condition, and the first transaction applies the same WHERE condition and gets the row inserted by the second transaction. • *TRANSACTION_NONE:* This isolation level indicates that transactions are not supported on this type of resource.	<session name="AccountServiceBean"> <resource-ref name="jdbc/Default" isolation-level="TRANSACTION_NONE"> </session>	Requires name attribute. To have any effect, also include the isolation-level attribute.

Extensions defined in META-INF/ibm-ejb-jar-ext-pme.xml file

The following tables list extension elements and attributes that must be placed in the META-INF/ibm-ejb-jar-ext-pme.xml file.

Table 68. Extensions defined in `META-INF/ibm-ejb-jar-ext-pme.xml`. Extensions defined in META-INF/ibm-ejb-jar-ext-pme.xml

Element or Attribute	Description	Example	Usage notes
<internationalization>	Element that might be used to declare the locale that might be used by the EJB type (locale of the caller or locale of the server).	<internationalization> <application> <ejb name="S01"/> <ejb name="S02"/> </application> <run-as-caller> <method type="LOCAL" name="getFoo" params="int"> <ejb name="C01"/> </method> </run-as-caller> <run-as-server> <method type="LOCAL" name="getBar" params="int"> <ejb name="C02"/> </method> </run-as-server> <run-as-specified name="North American English"> <locale lang="en" country="US" variant="foo"/> <locale lang="en" country="CA" variant="bar" /> <time-zone name="GMT"/> <method type="LOCAL" name="getFoo" params="int"> <ejb name="C03"/> </method> </run-as-specified> <run-as-specified name="North American French"> <locale lang="fr" country="US" variant="foo"/> <locale lang="fr" country="US" variant="bar" /> <time-zone name="GMT" /> <method type="LOCAL" name="getBar" params="int"> <ejb name="C04"/> </method> </run-as-specified> </internationalization>	For information about this extension, see Container internationalization attributes: WebSphere Application Server. Due to the complexity of this function, you might want to use a tool designed for WebSphere Application Server such as Rational Application Developer to produce the wanted extension file stanzas, then modify the XML file as wanted.
<activity-sessions>	Element that optionally declares the type of activity session management to be used on a designated session bean (BEAN or CONTAINER) and for container-managed activity sessions, the type of activity session behavior to be provided by the container.	<activity-sessions> <container-activity-session name="Foo" type="NOT_SUPPORTED"> <methodtype="HOME" name="findByPrimaryKey" params="int"> <ejb name="C01"/> </method> </container-activity-session> <./activity-sessions>	For information about this extension, see Setting EJB module ActivitySession deployment attributes. Due to the complexity of this function, you might want to use a tool designed for WebSphere Application Server such as Rational Application Developer
<app-profiles>	Element that optionally declares application profile settings for one or more EJB files.	<app-profiles> <defined-access-intent-policy name="foo"> <collection-scope type"SESSION"/> <optimistic-read/> <read-ahead-hint hint="foo.bar.baz"/> </defined-access-intent-policy> <run-as-task name="TestEJB1.ejbs.C01LocalHome.createjava.lang.Integer" type="RUN_AS_SPECIFIED_TASK"> <task name="/> <method type="LOCAL" name="getFoo" params="int"> <ejb name="C01"/> </method> </run-as-task> <ejb-component-extension ejb="C01"> <task name="SomeTask"/> </ejb-component-extension> </app-profiles>	Due to the complexity of this function, you might want to use a tool designed for WebSphere Application Server such as Rational Application Developer to produce the wanted extension file stanzas, then modify the XML file as wanted.

Legacy (XMI) bindings

Existing modules and applications can continue to use the legacy binding support provided in the product, therefore, the existing tools and wizards can be used to specify binding and extension information for applications and modules. Use of the legacy support is limited to EAR files and modules using J2EE 1.4-style XML deployment descriptors.

EJB modules that use a version 3.x XML deployment descriptor schema or do not have an XML deployment descriptor file must use either defaulted bindings and AutoLink, or user-specified XML binding files.

It is required that CMP entity beans always be packaged in a module with a 2.1 XML deployment descriptor schema version so that existing tools can be used to provide mappings, bindings, and extension support.

User-specified XML bindings

The default bindings for each interface and AutoLink reference resolution for each reference can be overridden by specifying bindings for the EJB module by creating a `META-INF/ibm-ejb-jar-bnd.xml` file.

The schema files that describe the format are located in the <WAS_HOME>/properties/schemas directory. This form of bindings specification can only be used for modules containing either no XML deployment descriptor or an EJB 3.x deployment descriptor.

Note: It is not required to specify all bindings. Any binding name or reference that is not defined uses the default bindings and AutoLink support.

Bindings can be specified for the following beans:

- Session beans using the <session> element.
- Message Driven beans using the <message-driven> element

The only attributes and subelements supported for the <session> element are:

- id attribute
- name attribute
- simple-binding-name attribute
- component-id attribute
- ejb-ref element
- resource-ref element and its attributes
- resource-env-ref element and its attributes
- message-destination-ref element and its attributes
- env-entry element
- data-source element

The only attributes and sub elements supported for the <message-driven> element are:

- id attribute
- name attribute
- jca-adapter attribute
- ejb-ref element and its attributes
- resource-ref element and its attributes
- resource-env-ref element and its attributes
- message-destination-ref element and its attributes
- env-entry element
- data-source element

EJB 3.x module packaging overview

This topic describes application packaging when you use Enterprise JavaBeans (EJB) 3.x beans.

Packaging applications that use EJB 3.x beans is similar to the assembly requirements for Java Platform, Enterprise Edition (Java EE) 1.4 applications: components are packaged into modules, and modules are packaged into application enterprise archive (EAR) files. The components and modules both have describing metadata provided in an Extensible Markup Language (XML) deployment descriptor. The EJB 3.x specifications support an additional method to describing metadata and for packaging persistence units.

The EAR file is a package file format similar to a .zip or .tar file format. The EAR file can be visualized as a collection of logical directories and files that are packaged together into a simple file. Each EAR file includes one or more Java EE module files, which can include the following modules:

- Java application archive (JAR) files for EJB modules. Java EE application client modules and utility class modules.
- Web application archive (WAR) files for web modules, or EJB content. The WAR file must be version 2.5 or later to contain EJB content.
- Other technology-specific modules such as resource application archive (RAR) files and other types of modules.

EJB modules without deployment descriptors

You can package EJB modules without a deployment descriptor if you are using EJB 3.x beans. To do this, you must create a JAR file or WAR file with metadata in an annotation which is located in the EJB component. EJB 3.x beans do not need an entry in the `ejb-jar.xml` file for metadata that you have defined through annotations.

With EJB 3.0, the default was to scan annotations during the installation of an EJB 3.0 module. For WebSphere Application Server, Version 8.5, the default is not to scan pre-Java EE 5 modules during the application installation or at server startup.

To preserve backward compatibility with both the Feature Pack for EJB 3.0 and the Feature Pack for Web Services, you have a choice whether to scan legacy web modules for additional metadata. A server level switch is defined for each feature pack scan behavior. If the default is not appropriate, the switch must be set on each server and administrative server that requires a change in the default. The switches are server custom properties com.ibm.websphere.webservices.UseWSFEP61ScanPolicy={true|false} and com.ibm.websphere.ejb.UseEJB61FEPScanPolicy={true|false}. To define these properties in the administrative console click **Application servers** > ***server name*** > **Process definition** > **Java Virtual Machine** > **Custom properties**.

EJB modules with deployment descriptors

You can continue to use EJB modules with deployment descriptors. Modules with deployment descriptors can support any EJB specification version level, including EJB 3.x, but generally these descriptors should reflect the implementation requirements of the components in the module.

An EJB module can have an EJB 3.x, 2.x, or 1.x deployment descriptor.

For EJB 2.x or 1.x deployment descriptors, it is assumed that the deployment descriptor contains the full metadata for the module, and no additional scanning of annotation metadata occurs.

The EJB container annotation scanning is performed on EJB modules that either have no deployment descriptor or have an `ejb-jar.xml` deployment descriptor at the EJB 3.0 schema level with the metadata-complete XML attribute set to false or omitted. See the Annotation scanning behavior section for the complete set of rules used by the server to determine if annotation scanning is performed.

Note: You cannot scan for component annotation metadata contained within shared libraries defined using the WebSphere Application Server system management shared library feature. However, content defined as a BLA Asset is scanned for annotation data.

Annotation scanning behavior

The server can inspect the class files in the module for annotation content. The server searches for annotation content that might define a component, a reference to a resource, or a particular behavior. For example, annotations might be used to define an EJB component, or to declare a reference to a data source that must be used by an EJB component, or to declare the transactional or security attributes that are associated with an EJB method. This inspection process is referred to as annotation scanning. If the class files in the module contain annotations that must be respected by the server, then the server must be configured so that annotation scanning occurs. If the class files in the module do not contain annotations, then for performance reasons you can configure the server so that annotation scanning does not occur.

The server uses the following criteria to determine whether it scans content for annotations:

- Whether the ejb-jar.xml deployment descriptor file exists
- Version of the ejb-jar.xml deployment descriptor, when it is exists
- Value of the metadata-complete setting in the ejb-jar.xml deployment descriptor, when it exists
- Version of the web.xml deployment descriptor, when the EJB content is packaged in a WAR module, and the ejb-jar.xml deployment descriptor does not exist
- Value of the metadata-complete setting in the web.xml deployment descriptor, when the EJB content is packaged in a WAR module, and the ejb-jar.xml deployment descriptor does not exist

The following tables indicate how the decision to scan, or not scan, annotations is made for EJB content that is packaged in an EJB JAR module or a WAR module.

Table 69. Annotation scanning for EJB content packaged in an EJB JAR module. Annotation scanning for EJB content packaged in an EJB JAR module

ejb-jar.xml	metadata-complete value in ejb-jar.xml	Are annotations scanned?
Exists, with a version of 2.x or earlier	NA	No
Exists, with a version of 3.x or later	true	No
Exists, with a version of 3.x or later	false (or omitted)	Yes
Does not exist	NA	Yes

Table 70. Annotation scanning for EJB content packaged in a WAR module. Annotation scanning for EJB content packaged in a WAR module

ejb-jar.xml file	metadata-complete value in ejb-jar.xml	web.xml file	metadata-complete value in web.xml	Are annotations scanned?
Exists, with a version of 3.x or later	true	NA	NA	No
Exists, with a version of 3.x or later	false (or omitted)	NA	NA	Yes
Exists, with a version of 2.x or earlier	NA	NA	NA	No
Does not exist	NA	Exists, with a version of 2.5 or later	true	No
Does not exist	NA	Exists, with a version of 2.5 or later	false (or omitted)	Yes
Does not exist	NA	Exists, with a version of 2.4 or earlier	NA	No
Does not exist	NA	Does not exist	NA	Yes

Note:

It is important to understand the distinction between the metadata-complete attribute of the ejb-jar element of the `ejb-jar.xml` deployment descriptor, and the metadata-complete install setting that may be specified during the application or module installation process.

The metadata-complete attribute of the ejb-jar element of the `ejb-jar.xml` file is an XML attribute. It is used by the server to determine if classes must be scanned for annotation data, as just described by the rules in the Annotation Scanning For EJB Content tables.

In contrast, the metadata-complete setting that may be specified at install time is used by the server to help generate the `ejb-jar.xml` file. If no `ejb-jar.xml` file exists in the module, and the metadata-complete install setting is assigned a value of true, then the server scans for annotation content and uses that to generate an `ejb-jar.xml` file, and then sets the metadata-complete XML attribute in that file to a value of true.

Persistence units

Persistence units, including the `persistence.xml` file and the classes associated with it, can be packaged in the module for which they are required. They can also be packaged in the separate utility JAR file that is packaged in the EAR file with its dependent module.

When a separate utility JAR file is packaged, it is necessary for the module that desires it to use the persistence units to declare a dependency on the utility JAR file using the typical MANIFEST. MF Class-Path: declarations. See the example scenario for this packaging method under the section in this topic called “Session facades used for persistence scenario”

Note: Packaging of persistence units contained within shared libraries defined using the WebSphere Application Server system management shared library feature is not supported at this time.

Application packaging

You can mix EJB 2.x and earlier beans with EJB 3.x beans in the same application. However, EJB 3.x beans are not recognized in EJB 2.x or EJB 1.x modules.

In the case that the EAR file only contains the JAR and web application archive (WAR) files, and no `application.xml` file, the product provides a default J2EE 1.4 deployment descriptor that is based on the following defaults that are outlined in the Java EE specification:

- The application name is assumed to be the name of the EAR file, but with the EAR file extension removed.
- Files that are ending in .war are assumed to be web modules. The context root of the web module is the name of the file that is relative to the root of the application package, but with the WAR file extension removed.
- Files that are ending in .jar that are not in the /lib directory, and that contain either an `ejb-jar.xml` file or at least one class that defines a @Stateful, @Stateless, @Singleton, or @MessageDriven annotation, are assumed to be EJB modules.
- Other JAR files that are not in the /lib directory are not assumed to be EJB modules.

If the application archive file contains an `application.xml` descriptor, processing occurs according to the directives in that descriptor.

AutoLink

AutoLink provides the ability to attempt to automatically resolve EJB references to components contained with an EAR file, without having to specify a JNDI binding name. This simplifies application deployment with large numbers of beans and references if they are unique and unambiguous.

Restriction: AutoLink should not be used for references to components deployed on a cluster.

JPA packaging

It is recommend that persistence units be packaged in separate JAR files to make them more accessible and reusable. These can be tested outside the container, with or without actual database persistence occurring. Persistence units can be included in stand-alone applications or into EAR files as utility JAR files. Because of the variety of use cases and potential performance issues when scanning large quantities of classes, it is recommended that the persistence unit defines the classes of the persistence units.

Session facades used for persistence scenario

A common pattern is to use session facades for persistence. Using session bean facades to drive JPA is supported. The EntityManager interface is not thread safe, therefore, servlets should never inject @PersistenceContext. Servlets must either use the facade pattern or use an EntityManagerFactory instance to create an EntityManager on each request.

It is recommended that JPA persistence units be defined in a separate JAR file, apart from the session bean facades. Not only is this a best practice that gives greater flexibility in sharing, it also avoids problems mixing JPA and non-JPA annotated classes.

Typically, a JAR file is created to hold the entity classes and the JPA `persistence.xml` definition and added to the EAR file as a utility JAR file. The EJB 3.x module adds a dependency on the JAR file by declaring it in the EJB 3.x module MANIFEST.MF. For example, if an EAR contains a `TradeApp.ear`, `TradeWeb.war`, `EJB3Trade.jar`, and `TradeInfo.jar` file, the `EJB3Trade.jar` file would have a MANIFEST.MF that looks like the following:

```
Manifest-Version: 1.0
Class-Path: TradeInfo.jar
```

The session facade in the `EJB3Trade.jar` file refers to JPA entity classes and persistence units in the `TradeInfo.jar` file. The web application defined in the `TradeWeb.war` file can do the same to work with the JPA entity objects as Data Transfer Objects flowing between the web and EJB container tiers.

Cross-tier and cross version session bean reference scenario

There are several ways to define and use references to EJB 3.x session beans. For EJB 3.x session to session, the @EJB injection target can be used. For cross-tier, for example, web application to EJB 3.x session, or cross-version, for example, EJB 2.1 session to EJB 3.x session, an XML deployment descriptor reference can be used to define ejb-refs and ejb-local-refs. There are two variations of these, depending on whether an EJB 3.x business interface is referred to, or a pre-EJB 3.x component-style interface that also defines an EJBLocalHome is referred to. Web applications and client applications can also use the @EJB annotation if the component being referenced can be resolved using autolink.

For migration scenarios where session beans are being converted from EJB 2.1 beans to EJB 3.x beans, the pattern is typically to edit the Session bean class, replace the implements SessionBean with implements the business interface, remove extends EJBLocalObject from the local interface and non-business throws clauses, and add the @Stateful @Local @LocalHome(<localhome>.class) or similar annotations. Existing ejb-refs and ejb-local-refs are bound to the new implementation of the session bean.

Note: The default binding name does change.

The previous scenario uses an EJB 2.1-style client pattern with an EJB 3.x-style session bean implementation. For a more current client style, the client-side can be cleaned up to look up the session bean business interface directly, rather than going through a home interface. In this case, it is not necessary to define the @LocalHome(<localhome>.class) annotation. You can use a variant definition of

ejb-ref and ejb-local-ref to do this. Use a `null` value for the local-home element value and bind the ejb-local-ref to the session bean's ejblocal: binding rather than the home binding. For example:

```
<ejb-local-ref id="EJBLocalRef_1154112538064">
 <description>com.ibm.persistence.ejb3.order.facadecom.ibm.persistence.ejb3.order.facade</description>
 <ejb-ref-name>ejb/OrderEJB</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home></local-home>
 <local>com.ibm.persistence.ejb3.order.facade.OrderProcessor</local>
</ejb-local-ref>
```

The client code also must be adjusted to do the appropriate casting for the object being looked up. In this case, the business interface instead of the home interface:

```
try {
 InitialContext ctx = new InitialContext();
 orderProcessor = (OrderProcessor)ctx.lookup("java:comp/env/ejb/OrderEJB");
}
catch(Exception e)  {
 e.printStackTrace(System.out);
 throw new ServletException(e);
}
```

Assembling EJB 3.x modules

An enterprise bean is a managed Java component that can be combined with other resources to create Java Enterprise Edition (Java EE) applications.

Before you begin

This topic assumes that you have created and unit tested an enterprise bean that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Assemble an EJB 3.x module to contain enterprise beans and related code artifacts. Group web components, client code, and resource adapter code in separate modules. After the EJB module is assembled, install it as a stand-alone application or combine it with other modules into an enterprise application.

To learn about how to assemble an EJB 3.x module with an assembly tool, see the Rational Application Developer documentation.

Rational Application Developer can be extended with additional plug-ins to provide development support specifically for Java Persistence API (JPA).

See the Eclipse open source project, Dali, for a plug-in that provides this extension. See the related links in this topic for the Dali JPA tools website.

Note: Issues and problems using this plug-in need to resolved through the Eclipse open source community.

What to do next

Deploy the EJB module.

Assembling EJB modules

An enterprise bean is a Java component that can be combined with other resources to create Java Platform Enterprise Edition (Java EE) applications. This topic describes assembling Enterprise JavaBeans (EJB) modules based on the EJB specifications.

Before you begin

This topic assumes that you have created and unit tested an enterprise bean file that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Assemble an EJB module to contain enterprise beans and related code artifacts. Group web components, client code, and resource adapter code in separate modules. After assembling an EJB module, you can install it as a stand-alone application or combine it with other modules into an enterprise application.

Use an assembly tool to assemble an EJB module in any of the following ways:
- Import an existing EJB module (EJB JAR file).
- Create a new EJB module.
- Copy code artifacts, such as entity beans, from one EJB module into a new EJB module.

For information on assembling EJB modules, refer to the online documentation or the information center for your assembly tool. The Rational Application Developer product provides supported assembly tools.

Procedure
- Assemble an EJB 2.x module. See the topic Assembling EJB 2.x modules.
- Assemble an EJB 3.x module. See the topic Assembling EJB 3.x modules.

What to do next

After you finish assembling your EJB module, you are ready to deploy your module.

EJB modules

An Enterprise JavaBeans (EJB) module is used to assemble one or more enterprise beans into a single deployable unit. An EJB module is stored in a standard Java archive (JAR) file.

An EJB module contains the following:
- One or more deployable enterprise beans.
- A deployment descriptor, stored in an Extensible Markup Language (XML) file. This file declares the contents of the module, defines the structure and external dependencies of the beans in the module, and describes how the beans are to be used at run time.

  It is not necessary to use XML deployment descriptors in EJB 3.x modules, although XML descriptors are supported. Instead of deployment descriptors, you can use annotations to provide component metadata.

You can deploy an EJB module as a stand-alone application, or combine it with other EJB modules or with web modules to create a Java application. An EJB module is installed and run in an enterprise bean container.

If you want to package an EJB 3.x module with a deployment descriptor, there are several ways to do it. You can package an EJB 3.x module with an EJB 3.x style session and/or message-driven beans exclusively; with an EJB 2.1 style session and/or message-driven beans exclusively, or a combination of 2.1 and 3.x style beans. The XML deployment descriptor must be a Version 3.x deployment descriptor. It is required that 2.1 entity beans are packaged in modules with 2.1 deployment descriptors.

EJB modules that contain EJB 3.x beans must be at the EJB 3.x specification level when running on the product. To set the EJB module to support EJB 3.x beans, you can set the ejb-jar.xml deployment descriptor level to 3.0 or 3.1, or you can make sure that the module does not contain an ejb-jar.xml

deployment descriptor. If the module level is EJB 2.1 or earlier, no EJB 3.x functions, including annotation scanning or resource injection is performed at run time.

For more information about packaging and deployment of EJB 3.x beans, see the topic EJB 3.x module packaging overview.

Local client views

The EJB specification only requires local client views to be supported for EJBs packaged within the same application. This includes local homes, local business interfaces, and the no-interface view. The product permits access to local client views to EJBs packaged within a separate application with some restrictions:

- The local interface and all parameter, return, and exception types used by the local interface must be visible to the class loader of both the calling application and the target EJB application. You can ensure this by either using a shared library associated with a server class loader or by using an isolated shared library associated with both applications. Read the Creating shared libraries topic for more information.
- When the target EJB application is stopped, any cached references to the EJB must be refreshed. You can complete one of the following actions:
  - Restart the calling application. The simplest solution is to restart the calling application whenever you restart a target EJB application on which it relies.
  - Obtain a new reference from JNDI. By default, JNDI lookups from the java namespace are cached, and the cache must either be disabled or cleared to obtain a new reference. Read the Developing applications that use JNDI topic for more information.

    EJB method invocations throw `com.ibm.websphere.ejbcontainer.EJBStoppedException` when the target EJB application has been stopped. If you have cached the EJB reference in an instance variable by using either @EJB injection or JNDI lookup, then you can catch this exception and refresh the EJB reference by performing a non-cached lookup.
  - Enable indirect local EJB proxies for the target EJB application. This causes the local EJB proxy to be refreshed automatically when the application is restarted. Enabling indirect local proxies causes some additional overhead for each EJB method invocation.

    You can enable indirect local proxies using, for example, an administrative console. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Custom properties** > **New**. Specify a name of `com.ibm.websphere.ejbcontainer.indirectLocalProxies` and a value of `true` for the custom property, then apply and save the changes.

EJB content in WAR modules

Use this topic to understand the packaging requirements of Enterprise JavaBeans (EJB) content in web application archive (WAR) modules.

Supported EJB content

Except for explicitly stated restrictions, the EJB function that is supported for beans packaged inside EJB Java archive (JAR) modules is also supported for beans packaged inside WAR modules. A bean that is packaged inside a WAR module can have the same behavior as a bean that is packaged inside an EJB JAR module.

All types of EJB 3.x beans are supported in WAR modules, and 2.x and 1.x session beans. See the EJB 3.1 specification for complete details.

Packaging mechanics

The rules for packaging EJB content in a WAR module are different from the rules for packaging EJB content in a JAR module.

The bean class files must be placed in one of two locations within the WAR module:

- Loosely in the WEB-INF/classes directory structure
- Within a JAR file that is placed in the WEB-INF/lib directory

For example, you might loosely place the bean class com.foo.MyBean in the WAR module at this location: WEB-INF/classes/com/foo/MyBean.class.

You might also place this bean class in the myJar.jar file, which is then placed in this location: WEB-INF/lib/myJar.jar.

A WAR module can have some bean code loosely placed in the WEB-INF/classes directory structure, and have other bean code inside JAR files in the WEB-INF/lib directory. It is also valid for a WAR module to have all the bean code in the WEB-INF/classes directory structure and nothing in the WEB-INF/lib directory, or all the bean code in JAR files in the WEB-INF/lib directory and nothing in WEB-INF/classes.

It is valid to have multiple JAR files in the WEB-INF/lib directory, all of which might contain bean code.

If the same bean class is loosely placed in the WEB-INF/classes directory structure and also placed in a JAR file in the WEB-INF/lib directory, then the instance of the class placed loosely in the WEB-INF/classes directory structure is loaded, and the instance placed in a JAR file in the WEB-INF/lib directory is ignored.

If the same bean class is placed in two different JAR files in the WEB-INF/lib directory, it is not known which instance of the class is loaded, and which instance is ignored. At run time, the server arbitrarily picks one class instance and loads it, and ignores the other class instance.

The EJB deployment descriptor files must be placed in the WEB-INF directory. This directory contains the ejb-jar.xml deployment descriptor, and any ibm-ejb-jar-ext and ibm-ejb-jar-bnd XML or XMI bindings and extensions files. EJB descriptor files that are located inside JAR files in the WEB-INF/lib directory are ignored. As with an EJB JAR module, there might be 0 or 1 instance of each EJB descriptor file. There cannot be multiple instances of any EJB descriptor file. This does not include the persistence.xml file, if one exists. Per the Java Persistence API specification, if a persistence.xml file is present, it must remain in a META-INF directory located in either the WEB-INF/classes directory of the WAR module or in a JAR file in the WEB-INF/lib directory of the WAR module. For example:

- WEB-INF/classes/META-INF/persistence.xml
- WEB-INF/lib/MyEntity.jar

The MyEntity.jar contains META-INF/persistence.xml.

If an ejb-jar.xml file is located in a JAR file in the WEB-INF/lib directory, a warning message displays. For example:

```
IWAE0068W The EJB deployment descriptor META-INF/ejb-jar.xml in the library archive
 foo.jar file is ignored. The product does not process the META-INF/ejb-jar.xml deployment
 descriptor in library archives. Move the META-INF/ejb-jar.xml deployment descriptor from the
 library archive to the WEB-INF directory in the WAR module.
```

A WAR module must be version 2.5 or later to contain EJB content. EJB content placed in a WAR module that is version 2.4 or earlier is ignored.

Note: If a WAR module is version 2.5 or later, the web metadata files containing bindings and extensions information must use the XML version of the files, not the XMI version.

Technical differences for enterprise beans that are packaged in a WAR file

The following list contains key technical differences that exist between beans that are packaged in a WAR module and beans that are packaged in an EJB JAR module:

- Shared component namespace

All components in a WAR module share a single component namespace. This means that each EJB component shares a single component namespace with all other EJB components in the WAR file, and any non-EJB components like servlets. In contrast, an EJB component that is packaged in an EJB JAR module has its own private component namespace, which is not shared with any other component.

The shared component namespace has important impacts. First, one component (EJB or non-EJB) might declare a reference, and a different component might search the component namespace for that reference. Second, references declared by one component might conflict with references declared by another component. In contrast, an EJB packaged in an EJB JAR module cannot look up in the component namespace a reference declared by a different EJB or non-EJB component, and it is impossible for a reference declared by the EJB to conflict in the component namespace with a reference declared by any other component, even if the references have the same name.

When using the shared namespace, it is valid for the same reference to be declared multiple times, as long as these reference declarations do not conflict with each other. If the reference declarations do not conflict, then the server behaves as if the reference had been declared exactly once.

If reference declarations do conflict, then an error message is emitted, and the application fails to start. A warning message is emitted for each conflicted reference. The warning message indicates the name of the conflicted reference, and the multiple values assigned to that reference. After all the warning messages are emitted, an exception is thrown.

- Location of EJB descriptor files

  The `ejb-jar.xml` deployment descriptor file, and any other descriptor file, must be placed in the `WEB-INF` directory of the WAR. Any instance of an EJB descriptor file elsewhere in the WAR, including in the `META-INF` directory of a JAR file in the `WEB-INF/lib` directory, is ignored.

- Determining if annotations are scanned

  The rules for determining whether to scan for annotations are different for EJB JAR and WAR modules. See the topic, EJB 3.x module packaging overview, for the complete set of rules.

- Class loading and visibility

  The most common usage pattern for EJB classes that are packaged in a WAR module is local method invocations from web components packaged within the same module. However, these EJB classes can also be accessed by remote method invocations or by clients in other modules. In these cases, it is important to understand the visibility rules of EJB classes that are packaged in a WAR module. Visibility rules are different when compared to EJB classes that are packaged in a JAR module.

  In the case of remote EJB method invocations, there are no visibility differences introduced by packaging the EJB classes in a WAR module. The EJBs are bound into the global namespace and can be looked up from, or injected into, components in other modules. The remote client must make method invocations with an appropriate stub class. Stub class generation is described in this topic under the section, "Stub generation".

  In the case of local EJB method invocations from components in other modules, there are visibility differences because the EJBs are packaged in a WAR module. These visibility differences occur because there are class loader implications that must be considered.

  The content that is packaged in all EJB JAR modules for the entire application is loaded by a single application class loader instance.

  In contrast, all content that is packaged in a WAR module is loaded on a class loader instance that is specific to that WAR module. The single application class loader instance that is used to load all the EJB JAR content, is the parent to each of the class loader instances that are used to load the WAR content.

  The visibility of a class is affected by the class loader instance that loaded it. A class loader instance can see classes loaded by itself, or by a parent class loader. However, a class loader cannot see a class loaded on a class loader that is not itself, nor one of its parents.

  As a result, classes loaded by a class loader specific to a WAR module can see classes in an EJB JAR module, but they cannot see classes in another WAR module. Classes in an EJB JAR module cannot

see classes in any WAR module. For example, if there is EJB content packaged inside EJB JAR module ejb3.jar, and there is also EJB content packaged inside the ejb1.jar file and the ejb2.jar file, then:

- If the ejb1.jar file and the ejb2.jar file are installed as EJB JAR modules, then the content inside the ejb1.jar file, ejb2.jar file, and ejb3.jar file is all loaded on the same class loader instance, which is also used to load any other EJB JAR modules in the application. In this case, the classes in all three JAR files can see each other, because they are all loaded by the same class loader instance.
- If the ejb1.jar file and the ejb2.jar file are both packaged inside the WEB-INF/lib directory of a WAR file, the content inside the ejb1.jar file and the ejb2.jar file is loaded by a single class loader instance. However, this class loader is not the same one used to load the content for the ejb3.jar file and any other EJB JAR in the application. In this case, the classes in the ejb1.jar file and the ejb2.jar file can see each other and can also see the classes in the ejb3.jar file. The classes in the ejb3.jar file cannot see the classes in the ejb1.jar file or the ejb2.jar file.
- If the ejb1.jar file is packaged inside the WEB-INF/lib directory of the firstWar.war file, and the ejb2.jar file is packaged inside the WEB-INF/lib directory of the secondWar.war file, the content in the ejb1.jar file is loaded on one class loader instance, the content in the ejb2.jar file is loaded on a second class loader instance, and the content in the ejb3.jar file and all other EJB JAR in the application is loaded on a third class loader instance. In this case, the classes in the ejb1.jar file and the ejb2.jar file cannot see each other, but they can see the classes in the ejb3.jar file. The classes in the ejb3.jar file cannot see the classes in either the ejb1.jar file or the ejb2.jar file.

One strategy to avoid these class loader complications is to package the EJB interface classes in a shared library.

Note: Do not package the same class, both an EJB JAR module and a WAR module, in the same application. Packaging the same class in multiple locations within the same application might result in confusion regarding which instance of the class is loaded and used at run time. This distinction can matter if the two .class files represent different versions of the class. To avoid this scenario, package the .class file in only one location or change the package structure of the class so that the fully qualified name of the class packaged inside the WAR module is different from the fully qualified name of the class packaged inside the EJB JAR module.

- Application profile extension

  The application profile extension is not supported for EJB classes that are packaged in WAR modules.

Stub generation

Remote access of EJB methods requires the use of client-side stub classes. For most client environments, the product runtime automatically generates the required stub classes. One exception is the thin client environment. For thin clients, the stub classes must be manually generated and packaged with the client.

Use the **createEJBStubs** tool to generate stubs when the EJB content is packaged in a WAR module, regardless of the EJB version.

See the topic, Create stubs command, for more information.

Note: When packaging EJB 2.1 classes in a WAR module, do not include any stub classes generated by the EJBDeploy tool. These stub classes are different than the stub classes automatically generated by the product run time and can cause failures. In most situations the automatically generated stub classes are sufficient. The exception is if a component in the web module must make a remote method invocation on an EJB 2.1 class packaged in another JAR module. In this case, the EJBDeploy generated stub class for the EJB in the other JAR module must be packaged in the WAR module.

EJB 2.x and 1.x content in a WAR module

Except for entity beans, EJB 2.x and 1.x content is supported in a WAR module.

A 2.x or 1.x module packaged inside a WAR file requires an `ejb-jar.xml` deployment descriptor at version 2.x or 1.x in the `WEB-INF` directory of the WAR module. If XMI bindings and extension files are present, you must also package these bindings and files in the `WEB-INF` directory of a WAR module.

Session beans and message driven beans that you implement according to the 2.x or 1.x coding style can be packaged inside a WAR module.

Both bean managed persistence (BMP) and container managed persistence (CMP) entity beans are not supported in a WAR module.

The application profiling and access intent services are not supported in a WAR module. Session beans found in a WAR module cannot access application profiling tasks.

EJB content that you implement according to both the 3.x coding style and the 2.x and 1.x coding styles, can be packaged together in a single WAR module. However, in this case, you must declare any bindings and extensions information with the XML version of the files, not the XMI version.

Moving existing EJB content from EJB JAR modules into WAR modules

One approach is to put the existing EJB JAR file in the `WEB-INF/lib` directory of the WAR file. Then, remove the descriptor files from the `META-INF` directory of the JAR file and place them in the `WEB-INF` directory of the WAR file.

A second approach is to put the class files from the EJB JAR file in the correct location under the `WEB-INF/classes` directory in the WAR module. Then, remove the descriptor files from the `META-INF` directory of the JAR file and place them into the `WEB-INF` directory of the WAR file.

If multiple EJB JAR modules are moved into a single WAR module, you must merge the contents of each of the descriptor files previously found in the `META-INF` directories of the EJB JAR modules into the single version of the descriptor files that are now placed in the `WEB-INF` directory of the WAR file. Examples of descriptor files that may be merged include, but are not limited to, `ejb-jar.xml`, `ibm-ejb-jar-bnd.xml`, `ibm-ejb-jar-ext.xml`, and `ibm-ejb-jar-ext-pme.xml`.

You must inspect the references declared by the various components in the WAR module, both EJB and non-EJB, to ensure that they do not conflict with each other, since everything in the WAR module shares a single component name space.

You must modify the bindings and extension XMI files that are moved from an EJB JAR module to a WAR module in several places to remove references to `META-INF/ejb-jar.xml` and replace them with `WEB-INF/ejb-jar.xml`.

EJB function that is supported in EJB JAR modules, but not in WAR modules

The following EJB function is not supported in WAR modules:

- BMP and CMP entity beans
- Pre EJB 3.1 style startup beans

  Attention: Singleton startup beans defined by EJB 3.1 are supported.

If an entity bean is placed in a WAR module, error messages display, and the application fails to start. For example, the foo entity bean is placed in the foo.war module in the FooApp application. This results in the following messages:

```
CWMDF0025E: Entity beans in EJB web application archive (WAR) modules are not allowed,
 per the EJB 3.1 specification.
WSVR0039E: Unable to start EJB JAR, foo.war: Entity beans in EJB web application archive
 (WAR) modules are not allowed, per the EJB 3.1 specification. The foo bean in the foo.war module
 must be moved to a stand-alone EJB module. Examine the log to see a full list of invalid entity
 beans in a WAR module.
```

EJB 3.x module packaging overview

This topic describes application packaging when you use Enterprise JavaBeans (EJB) 3.x beans.

Packaging applications that use EJB 3.x beans is similar to the assembly requirements for Java Platform, Enterprise Edition (Java EE) 1.4 applications: components are packaged into modules, and modules are packaged into application enterprise archive (EAR) files. The components and modules both have describing metadata provided in an Extensible Markup Language (XML) deployment descriptor. The EJB 3.x specifications support an additional method to describing metadata and for packaging persistence units.

The EAR file is a package file format similar to a .zip or .tar file format. The EAR file can be visualized as a collection of logical directories and files that are packaged together into a simple file. Each EAR file includes one or more Java EE module files, which can include the following modules:

- Java application archive (JAR) files for EJB modules. Java EE application client modules and utility class modules.
- Web application archive (WAR) files for web modules, or EJB content. The WAR file must be version 2.5 or later to contain EJB content.
- Other technology-specific modules such as resource application archive (RAR) files and other types of modules.

EJB modules without deployment descriptors

You can package EJB modules without a deployment descriptor if you are using EJB 3.x beans. To do this, you must create a JAR file or WAR file with metadata in an annotation which is located in the EJB component. EJB 3.x beans do not need an entry in the `ejb-jar.xml` file for metadata that you have defined through annotations.

With EJB 3.0, the default was to scan annotations during the installation of an EJB 3.0 module. For WebSphere Application Server, Version 8.5, the default is not to scan pre-Java EE 5 modules during the application installation or at server startup.

To preserve backward compatibility with both the Feature Pack for EJB 3.0 and the Feature Pack for Web Services, you have a choice whether to scan legacy web modules for additional metadata. A server level switch is defined for each feature pack scan behavior. If the default is not appropriate, the switch must be set on each server and administrative server that requires a change in the default. The switches are server custom properties com.ibm.websphere.webservices.UseWSFEP61ScanPolicy={true|false} and com.ibm.websphere.ejb.UseEJB61FEPScanPolicy={true|false}. To define these properties in the administrative console click **Application servers** > ***server name*** > **Process definition** > **Java Virtual Machine** > **Custom properties**.

EJB modules with deployment descriptors

You can continue to use EJB modules with deployment descriptors. Modules with deployment descriptors can support any EJB specification version level, including EJB 3.x, but generally these descriptors should reflect the implementation requirements of the components in the module.

An EJB module can have an EJB 3.x, 2.x, or 1.x deployment descriptor.

For EJB 2.x or 1.x deployment descriptors, it is assumed that the deployment descriptor contains the full metadata for the module, and no additional scanning of annotation metadata occurs.

The EJB container annotation scanning is performed on EJB modules that either have no deployment descriptor or have an `ejb-jar.xml` deployment descriptor at the EJB 3.0 schema level with the metadata-complete XML attribute set to false or omitted. See the Annotation scanning behavior section for the complete set of rules used by the server to determine if annotation scanning is performed.

Note: You cannot scan for component annotation metadata contained within shared libraries defined using the WebSphere Application Server system management shared library feature. However, content defined as a BLA Asset is scanned for annotation data.

Annotation scanning behavior

The server can inspect the class files in the module for annotation content. The server searches for annotation content that might define a component, a reference to a resource, or a particular behavior. For example, annotations might be used to define an EJB component, or to declare a reference to a data source that must be used by an EJB component, or to declare the transactional or security attributes that are associated with an EJB method. This inspection process is referred to as annotation scanning. If the class files in the module contain annotations that must be respected by the server, then the server must be configured so that annotation scanning occurs. If the class files in the module do not contain annotations, then for performance reasons you can configure the server so that annotation scanning does not occur.

The server uses the following criteria to determine whether it scans content for annotations:

- Whether the ejb-jar.xml deployment descriptor file exists
- Version of the ejb-jar.xml deployment descriptor, when it is exists
- Value of the metadata-complete setting in the ejb-jar.xml deployment descriptor, when it exists
- Version of the web.xml deployment descriptor, when the EJB content is packaged in a WAR module, and the ejb-jar.xml deployment descriptor does not exist
- Value of the metadata-complete setting in the web.xml deployment descriptor, when the EJB content is packaged in a WAR module, and the ejb-jar.xml deployment descriptor does not exist

The following tables indicate how the decision to scan, or not scan, annotations is made for EJB content that is packaged in an EJB JAR module or a WAR module.

Table 71. Annotation scanning for EJB content packaged in an EJB JAR module. Annotation scanning for EJB content packaged in an EJB JAR module

ejb-jar.xml	metadata-complete value in ejb-jar.xml	Are annotations scanned?
Exists, with a version of 2.x or earlier	NA	No
Exists, with a version of 3.x or later	true	No
Exists, with a version of 3.x or later	false (or omitted)	Yes
Does not exist	NA	Yes

Table 72. Annotation scanning for EJB content packaged in a WAR module. Annotation scanning for EJB content packaged in a WAR module

ejb-jar.xml file	metadata-complete value in ejb-jar.xml	web.xml file	metadata-complete value in web.xml	Are annotations scanned?
Exists, with a version of 3.x or later	true	NA	NA	No
Exists, with a version of 3.x or later	false (or omitted)	NA	NA	Yes
Exists, with a version of 2.x or earlier	NA	NA	NA	No
Does not exist	NA	Exists, with a version of 2.5 or later	true	No
Does not exist	NA	Exists, with a version of 2.5 or later	false (or omitted)	Yes

Table 72. Annotation scanning for EJB content packaged in a WAR module (continued). Annotation scanning for EJB content packaged in a WAR module

ejb-jar.xml file	metadata-complete value in ejb-jar.xml	web.xml file	metadata-complete value in web.xml	Are annotations scanned?
Does not exist	NA	Exists, with a version of 2.4 or earlier	NA	No
Does not exist	NA	Does not exist	NA	Yes

Note:

It is important to understand the distinction between the metadata-complete attribute of the ejb-jar element of the `ejb-jar.xml` deployment descriptor, and the metadata-complete install setting that may be specified during the application or module installation process.

The metadata-complete attribute of the ejb-jar element of the `ejb-jar.xml` file is an XML attribute. It is used by the server to determine if classes must be scanned for annotation data, as just described by the rules in the Annotation Scanning For EJB Content tables.

In contrast, the metadata-complete setting that may be specified at install time is used by the server to help generate the `ejb-jar.xml` file. If no `ejb-jar.xml` file exists in the module, and the metadata-complete install setting is assigned a value of true, then the server scans for annotation content and uses that to generate an `ejb-jar.xml` file, and then sets the metadata-complete XML attribute in that file to a value of true.

Persistence units

Persistence units, including the `persistence.xml` file and the classes associated with it, can be packaged in the module for which they are required. They can also be packaged in the separate utility JAR file that is packaged in the EAR file with its dependent module.

When a separate utility JAR file is packaged, it is necessary for the module that desires it to use the persistence units to declare a dependency on the utility JAR file using the typical MANIFEST. MF Class-Path: declarations. See the example scenario for this packaging method under the section in this topic called “Session facades used for persistence scenario”

Note: Packaging of persistence units contained within shared libraries defined using the WebSphere Application Server system management shared library feature is not supported at this time.

Application packaging

You can mix EJB 2.x and earlier beans with EJB 3.x beans in the same application. However, EJB 3.x beans are not recognized in EJB 2.x or EJB 1.x modules.

In the case that the EAR file only contains the JAR and web application archive (WAR) files, and no `application.xml` file, the product provides a default J2EE 1.4 deployment descriptor that is based on the following defaults that are outlined in the Java EE specification:

- The application name is assumed to be the name of the EAR file, but with the EAR file extension removed.
- Files that are ending in .war are assumed to be web modules. The context root of the web module is the name of the file that is relative to the root of the application package, but with the WAR file extension removed.
- Files that are ending in .jar that are not in the /lib directory, and that contain either an `ejb-jar.xml` file or at least one class that defines a @Stateful, @Stateless, @Singleton, or @MessageDriven annotation, are assumed to be EJB modules.
- Other JAR files that are not in the /lib directory are not assumed to be EJB modules.

If the application archive file contains an `application.xml` descriptor, processing occurs according to the directives in that descriptor.

AutoLink

AutoLink provides the ability to attempt to automatically resolve EJB references to components contained with an EAR file, without having to specify a JNDI binding name. This simplifies application deployment with large numbers of beans and references if they are unique and unambiguous.

Restriction: AutoLink should not be used for references to components deployed on a cluster.

JPA packaging

It is recommend that persistence units be packaged in separate JAR files to make them more accessible and reusable. These can be tested outside the container, with or without actual database persistence occurring. Persistence units can be included in stand-alone applications or into EAR files as utility JAR files. Because of the variety of use cases and potential performance issues when scanning large quantities of classes, it is recommended that the persistence unit defines the classes of the persistence units.

Session facades used for persistence scenario

A common pattern is to use session facades for persistence. Using session bean facades to drive JPA is supported. The EntityManager interface is not thread safe, therefore, servlets should never inject @PersistenceContext. Servlets must either use the facade pattern or use an EntityManagerFactory instance to create an EntityManager on each request.

It is recommended that JPA persistence units be defined in a separate JAR file, apart from the session bean facades. Not only is this a best practice that gives greater flexibility in sharing, it also avoids problems mixing JPA and non-JPA annotated classes.

Typically, a JAR file is created to hold the entity classes and the JPA `persistence.xml` definition and added to the EAR file as a utility JAR file. The EJB 3.x module adds a dependency on the JAR file by declaring it in the EJB 3.x module MANIFEST.MF. For example, if an EAR contains a `TradeApp.ear`, `TradeWeb.war`, `EJB3Trade.jar`, and `TradeInfo.jar` file, the `EJB3Trade.jar` file would have a MANIFEST.MF that looks like the following:

```
Manifest-Version: 1.0
Class-Path: TradeInfo.jar
```

The session facade in the `EJB3Trade.jar` file refers to JPA entity classes and persistence units in the `TradeInfo.jar` file. The web application defined in the `TradeWeb.war` file can do the same to work with the JPA entity objects as Data Transfer Objects flowing between the web and EJB container tiers.

Cross-tier and cross version session bean reference scenario

There are several ways to define and use references to EJB 3.x session beans. For EJB 3.x session to session, the @EJB injection target can be used. For cross-tier, for example, web application to EJB 3.x session, or cross-version, for example, EJB 2.1 session to EJB 3.x session, an XML deployment descriptor reference can be used to define ejb-refs and ejb-local-refs. There are two variations of these, depending on whether an EJB 3.x business interface is referred to, or a pre-EJB 3.x component-style interface that also defines an EJBLocalHome is referred to. Web applications and client applications can also use the @EJB annotation if the component being referenced can be resolved using autolink.

For migration scenarios where session beans are being converted from EJB 2.1 beans to EJB 3.x beans, the pattern is typically to edit the Session bean class, replace the implements SessionBean with implements the business interface, remove extends EJBLocalObject from the local interface and

non-business throws clauses, and add the @Stateful @Local @LocalHome(<localhome>.class) or similar annotations. Existing ejb-refs and ejb-local-refs are bound to the new implementation of the session bean.

Note: The default binding name does change.

The previous scenario uses an EJB 2.1-style client pattern with an EJB 3.x-style session bean implementation. For a more current client style, the client-side can be cleaned up to look up the session bean business interface directly, rather than going through a home interface. In this case, it is not necessary to define the @LocalHome(<localhome>.class) annotation. You can use a variant definition of ejb-ref and ejb-local-ref to do this. Use a `null` value for the local-home element value and bind the ejb-local-ref to the session bean's ejblocal: binding rather than the home binding. For example:

```
<ejb-local-ref id="EJBLocalRef_1154112538064">
 <description>com.ibm.persistence.ejb3.order.facadecom.ibm.persistence.ejb3.order.facade</description>
 <ejb-ref-name>ejb/OrderEJB</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home></local-home>
 <local>com.ibm.persistence.ejb3.order.facade.OrderProcessor</local>
</ejb-local-ref>
```

The client code also must be adjusted to do the appropriate casting for the object being looked up. In this case, the business interface instead of the home interface:

```
try {
 InitialContext ctx = new InitialContext();
 orderProcessor = (OrderProcessor)ctx.lookup("java:comp/env/ejb/OrderEJB");
}
catch(Exception e)  {
 e.printStackTrace(System.out);
 throw new ServletException(e);
}
```

Defining container transactions for EJB modules

Container transaction properties specify how an Enterprise JavaBeans (EJB) container is to manage transaction scopes for the enterprise bean method invocations.

About this task

A transaction attribute is mapped to one or more methods. Some container transaction settings are not available for all enterprise beans. Also, some methods are not available for particular transaction settings and beans. These rules have been implemented in the Add Container Transaction wizard based on the EJB specification.

To complete this task see the topic, Defining container transactions for EJB modules, in the assembly tool information center.

References in application deployment descriptor files

References are logical names used to locate external resources for enterprise applications. References are defined in the application's deployment descriptor file. At deployment, the references are bound to the physical location (global Java Naming and Directory Interface (JNDI) name) of the resource in the target operational environment.

This product supports the following types of references:

- An Enterprise JavaBeans (EJB) reference is a logical name used to locate the home interface of an enterprise bean.
- A resource reference is a logical name used to locate a connection factory object.

These objects define connections to external resources such as databases and messaging systems. The container makes references available in a JNDI naming subcontext. By convention, references are organized as follows:

- EJB references are made available in the java:comp/env/ejb subcontext.
- Resource references are made available as follows:
  - JDBC data source references are declared in the java:comp/env/jdbc subcontext.
  - JMS connection factories are declared in the java:comp/env/jms subcontext.
  - Mail connection factories are declared in the java:comp/env/mail subcontext.
  - URL connection factories are declared in the java:comp/env/url subcontext.

EJB references

Use this page to view and modify the Enterprise JavaBeans (EJB) references to the enterprise beans. References are logical names used to locate external resources for enterprise applications. References are defined in the application's deployment descriptor file. At deployment, the references are bound to the physical location (global Java Naming and Directory Interface (JNDI) name) of the resource in the target operational environment.

If your application defines EJB references, for **Map EJB references to beans**, specify JNDI names for enterprise beans that represent the logical names that are specified in EJB references. Each EJB reference defined in the application must be bound to an EJB file before clicking Finish in the Summary panel.

If the EJB reference is from an EJB 3.x, Web 2.4, Web 2.5, or Client 5.0 module, the JNDI name is optional. If the **Allow EJB reference targets to resolved automatically** option is enabled, the JNDI name is optional for all modules. The runtime provides a container default or automatically resolves the EJB reference if a binding is not provided.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > **application_name** > **EJB references**.

Values are displayed for Lookup name and EJB Link if they are configured in the application. Only one of these values is allowed. If both are set, the value must be overridden by a target resource JNDI name.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Module

Specifies the name of the Enterprise JavaBeans module used by your application.

Bean

Specifies the name of an enterprise bean that is contained by the module.

URI

Specifies location of the module relative to the root of the application EAR file.

Resource Reference

Specifies the name of the EJB reference that is used in the enterprise bean, if applicable, and declared in the deployment descriptor of the application module.

Class

Specifies the name of a Java class associated with this enterprise bean.

Target Resource JNDI Name

Specifies the JNDI name of the enterprise bean.

This is a data entry field. To modify the JNDI name bound to this bean, type the new name in this field, then select **OK**.

Information	Value
Data type	String

EJB JNDI names for beans

Use this page to view and modify the Java Naming and Directory Interface (JNDI) names of non-message-driven enterprise beans in your application or module.

If your application uses Enterprise JavaBeans (EJB) 2.1 and earlier modules, on the Provide JNDI names for beans panel, specify a JNDI name for each enterprise bean in every EJB 2.1 and earlier module. You must specify a JNDI name for every EJB 2.1 and earlier enterprise bean defined in the application. For example, for the EJB module MyBean.jar, specify MyBean.

The JNDI name for an EJB module can be used for both EJB 3.x modules and pre-EJB 3.0 modules. For a pre-EJB 3.0 module, you need to provide a JNDI name for the bean. For an EJB 3.x module, you have three options

- Provide no JNDI names at all
- Select the radio button to provide a JNDI name for the bean, or
- Select the radio button to provide local or remote home JNDI names.

If no JNDI name is provided, the run time provides a default value. If JNDI name for the bean is provided, you cannot provide any JNDI name for business interface in the Provide JNDI names for business interfaces panel.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > **application** > **EJB JNDI names**.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Module

Specifies the name of the Enterprise JavaBeans module used by your application.

Bean

Specifies the name of an enterprise bean that is contained by the module.

URI

The Uniform Resource Identifier (URI) specifies the location of the module archive relative to the root of the application EAR.

Target Resource JNDI name

Specifies the Java Naming and Directory Interface (JNDI) name of the enterprise bean.

This is a data entry field. To modify the JNDI name bound to this bean, type the new name in this field, then select **OK**.

Information	Value
Data type	String

Bind EJB business settings

Use this administrative console page to specify Java Naming and Directory (JNDI) name bindings for each enterprise bean with a business interface in an EJB module. Each enterprise bean with a business interface in an EJB module must be bound to a JNDI name. For any business interface that does not provide a JNDI name, or if its bean does not provide a JNDI name, a default binding name is provided. If its bean provides a JNDI name, the default JNDI name for the business interface is provided on top of its bean JNDI name by appending the package-qualified class name of the interface.

If you specify the JNDI name for a bean in the Provide JNDI names for beans page, do not specify any business interface JNDI name in this page for the same bean. If you do not specify the JNDI name for a bean in the Provide JNDI names for beans page, you can optionally specify a business interface JNDI name. If you do not specify a business interface JNDI name, the run time provides a container default.

To view this page in the administrative console, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Bind EJB business**.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

Module

Specifies the EJB module that contains the enterprise beans that bind to the JNDI name.

Bean

Specifies the enterprise bean that binds to the JNDI name.

URI

The Uniform Resource Identifier (URI) specifies the location of the module archive relative to the root of the application EAR.

Business Interface

Specifies the enterprise bean business interface in an EJB module.

For a no-interface view, the business interface value is an empty string ("").

JNDI Name

Specifies the JNDI name associated with the enterprise bean business interface in an EJB module.

Developing EJB 2.x entity beans that use access intents

Using the AccessIntent API

This task describes how to programmatically retrieve and call the AccessIntent API during the execution of bean managed persistence (BMP) entity bean methods.

Procedure

1. Look up the access intent service from the namespace. For example:

```
InitialContext ic = new InitialContext();
AccessIntentService aiService = ic.lookup("java:comp/websphere/AppProfile/AccessIntentService");
```

2. From a method of the remote or local component interface of the BMP, get the current AccessIntent object using the javax.ejb.EntityContext. This object is passed to the BMP when the container calls the setEntityContext method. Assume the EntityContext was stored in a variable named myEntityCtx. For example:

```
AccessIntent ai = aiService.getAccessIntent (myEntityCtx);
```

3. Use the get() methods of AccessIntent interface to obtain the wanted information. For example:

```
int concurrency = ai.getConcurrencyControl();
int accessType = ai.getAccessType();
if ( (concurrency == AccessIntent.CONCURRENCY_CONTROL_PESSIMISTIC)
   && (accessType == AccessIntent.ACCESS_TYPE_UPDATE) ) {
     int exclusive = ai.getPessimisticUpdateLockHint();
     // . . .
}
// . . .
```

   For a detailed example of the use of the AccessIntent API, see the topic Example: Using IBM extended APIs to share connections between CMP beans and BMP beans..

Results

The access intent object reference retrieved from the java:comp lookup is current for the duration of the method in which the reference was looked up. Depending on how you configured the application profile, subsequent calls of the same method might not retrieve the same access intent reference. You can only look up the object reference during the call of a BMP entity bean method; the reference does not exist during a request on a container managed persistence (CMP) entity bean. Therefore, do not cache access intent object references beyond, or used outside of, the scope of the execution of any given BMP method.

AccessIntent interface

The AccessIntent interface is available to bean-managed persistence (BMP) entity beans.

A BMP entity bean can get and use an instance of the AccessIntent interface. For more information see “Using the AccessIntent API” on page 478.

AccessIntent interface

```
package com.ibm.websphere.appprofile.accessintent;

/**
* This interface defines the essential access intents
* available at run time.
*/
public interface AccessIntent {

/**
* Returns the concurrency control intent, which indicates
* the application prefers either pessimistic or optimistic
* concurrency control when accessing the current component
* in the context of the current transaction.
*/
public int getConcurrencyControl();
public final int CONCURRENCY_CONTROL_PESSIMISTIC = 1;
public final int CONCURRENCY_CONTROL_OPTIMISTIC = 2;

/**
* Returns access type intent, which indicates the application
* intends either update or read access of the current component
* in the context of the current transaction.
*/
public int getAccessType();
public final int ACCESS_TYPE_UPDATE= 1;
public final int ACCESS_TYPE_READ = 2;

/**
* Returns an integer value that indicates that the run time should
* assume that there will be no collision on retrieved rows.
*/
public int getPessimisticUpdateLockHint();
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NOCOLLISION = 1;
```

```
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_WEAKEST_LOCK_AT_LOAD = 2;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NONE = 3;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_EXCLUSIVE = 4;

/*
* Returns an integer value that indicates that the run time should
* assume that there will be collisions on retrieved rows.
*/
public int getPessimisticUpdateLockHint();
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NOCOLLISION = 1;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_WEAKEST_LOCK_AT_LOAD = 2;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_NONE = 3;
public final static int PESSIMISTIC_UPDATE_LOCK_HINT_EXCLUSIVE = 4;


/**
* Returns the collection access intent, which indicates the
* application intends to access the objects returned by the
* currently executing finder in either serial or random fashion.
*/
public int getCollectionAccess();
public final int COLLECTION_ACCESS_RANDOM = 1;
public final int COLLECTION_ACCESS_SERIAL = 2;

/**
* Returns the collection scope, which indicates the maximum
* lifespan of a lazy collection.
*/
public int getCollectionScope();
public final int COLLECTION_SCOPE_TRANSACTION = 1;
public final int COLLECTION_SCOPE_ACTIVITYSESSION = 2;
public final int COLLECTION_SCOPE_TIMEOUT = 3;

/**
* Returns the timeout value in seconds when collectionScope is Timeout.
*/
public int getCollectionTimeout();

/**
* Returns the number of elements the application requests be contained
* in each segment of the element collection returned by the currently
* executing finder.
*/
public int getCollectionIncrement();

/**
* Returns the ReadAheadHint requested by the application for the currently
* executing finder.
*/
public ReadAheadHint getReadAheadHint();

/**
* Returns the number of elements the application requests be contained in
* each segment of a query made on a database.
*/
public int getResourceManagerPreFetchIncrement();

}
```

Assembling access intents to EJB 2.x entity beans

Applying access intent policies to beans

You can apply an access intent policy to an application's entity beans through the assembly tool.

About this task

Container-managed persistence (CMP) developers can use *access intent* to provide hints on how the application server run time should manage the details of persistence without having to explicitly manage any of the persistence logic from within their application.

Using the access intent service is also an option for programmers who develop bean-managed persistence (BMP) entity beans. Because the only meaningful difference between BMP and CMP components is the mechanism that provides the persistence logic, BMP beans leverage access intent hints in the same manner as the EJB container manages accent intent for CMP beans. This ability becomes especially important when BMP entities and CMP entities want to share connections. BMP beans configured with the same concurrency as the CMP beans and implemented to the same isolation level mapping as the CMP can share connections.

Developers can apply access intent policies to BMP entity beans as well as to CMP entity beans. It is expected that BMP developers use only those access intent attributes that are important to a particular BMP bean. The access intent service interface is bound into the *java:comp namespace* for each particular BMP bean. The access intent policy retrieved from the access intent service is current from the time that the *ejbLoad* process is called until the time that the *ejbStore* process completes its invocation.

Note: This is the preferred technique to define access intent policies. Method-level access intent is deprecated in Version 6.0.

Procedure

1. Start an assembly tool.
2. Optional: Open the Java EE perspective to work with Java EE projects. Click **Window > Open Perspective > Other > Java EE**.
3. Optional: Open the Project Explorer view. Click **Window > Show View > Project Explorer**. Another helpful view is the Navigator view (**Window > Show View > Navigator**).
4. Create a new application EAR file or edit an existing one.

   For example, to change attributes of an existing application, use the import wizard to import an EAR file. To start the import wizard:

   a. Select **File > Import > EAR file > Next**
   b. Select the EAR file.
   c. Create a WebSphere Application Server v6.0 type of Server Runtime. Select **New** to open the New Server Runtime Wizard and follow the instructions.
   d. In the *Target server* field, select *WebSphere Application Server v6.0* type of Server Runtime.
   e. Select **Finish**
5. In the Project Explorer view of the J2EE perspective, right-click **Deployment Descriptor: EJB Module Name** under the EJB module for the bean instance, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the EJB project is displayed in the property pane.
6. Select the **Access** tab.
7. In the **Access Intent for Entities 2.x (Bean Level)** panel, select the name of the bean.
8. On the right side of the **Access Intent for Entities 2.x (Method Level)** panel, select **Add**. The **Add Access Intent** panel displays.
9. In the **Access intent name** field, select wsPessimisticUpdate from the drop-down list.
10. Optional: Enter a **Description** to help you remember what this policy does.
11. Optional: Change the **Persistence Option** setting
12. Click **Finish**. The access intent policy for the entity bean is shown in the **Access Intent for Entities 2.x (Bean Level)** panel

Configuring read-read consistency checking with an assembly tool

Read-read consistency checking only applies to LifeTimeInCache beans whose data is read from another transaction. The product checks that the data is consistent with the data store, and ensures that it is not updated after checking for access intents that are *repeatable read* (RR).

About this task

For access intents that are *read committed* (RC), the product checks that the data is consistent at the point of checking; it does **not** guarantee that the data does not change after the checking. This makes the behavior of the LifeTimeInCache bean the same as beans that are non-LifeTimeInCache.

To perform this checking, you need to configure CMP entity beans with read-read consistency checking. You can do this using an assembly tool. To learn how to complete this task see the topic, Adding bean-level access intent for entity beans 2.x in the assembly tool information center.

Example: Read-read consistency checking

Read-read consistency checking only applies to LifeTimeInCache beans whose data is read from another transaction.

Usage scenario

For the access intents that are for *repeatable read* (RR), this means the product checks that the data is consistent with that in the data store and ensures that no one updates it after the checking. For the Access Intents that are for *read committed* (RC), this means the product checks that the data is consistent at the point of checking, but it does **not** guarantee that the data does not change after the checking. This makes the behavior of the LifeTimeInCache bean the same as non-LifeTimeInCache beans.

You have three options for setting consistency checking, as shown in the following scenarios concerning the calculation of interest in "Ann's" bank account. In each case, the data store is shared by this Enterprise JavaBeans (EJB) container managed persistence (CMP) application to calculate the interest and other applications, such as EJB bean managed persistence (BMP) , Java Database Connectivity (JDBC), or legacy applications. Also in each case, the EJB account is configured as a *long-lifetime* bean.

NONE

- The server is started.
- User 1 in Transaction 1 calls Account.findByPrimaryKey("10001"), account data for Ann is read from the database, with a balance of $100.
- Ann’s record is cached by the persistence manager (PM) on the server.
- User 2 writes a JDBC call and changes the balance to $120.
- User 3 in Transaction 2 calls Account.findByPrimaryKey() for account "10001", Ann’s data is read from cache, with a balance of $100.
- Calculate Ann’s interest, but the result might not be correct because of the data integrity issue.

Read-read checking AT_TRAN_BEGIN

- The server is started.
- User 1 in Transaction 1 calls Account.findByPrimaryKey("10001"), account data for Ann is read from the database, with a balance of $100.
- Ann’s record is cached by the persistence manager (PM) on the server.
- User 2 writes a JDBC call and changes the balance to $120.
- User 3 in Transaction 2 calls Account.findByPrimaryKey() for account "10001", Ann’s data is read from cache, with a balance of $100.
- PM performs read-read check on Ann's account and finds that the balance of 100 is changed. It issues a database query to retrieve balance of $120, and Ann’s data in the cache is refreshed.

- Calculate Ann's interest, proceed with the transaction because data integrity is protected.

Read-read checking AT_TRAN_END

- The server is started.
- User 1 in Transaction 1 calls Account.findByPrimaryKey("10001"), account data for Ann is read from the database, with a balance of $100.
- Ann's record is cached by the persistence manager (PM) on the server.
- User 2 writes a JDBC call and changes the balance to $120.
- User 3 in Transaction 2 calls Account.findByPrimaryKey() for account "10001", Ann's data is read from database, with balance of $100.
- Calculate Ann's interest.
- During end of transaction 2, PM performs read-read check on Ann's account and finds that the balance of 100 is changed.
- PM rolls back the transaction and invalidates the cache. The transaction fails and again data integrity is protected.

Access intent service

Access intent is an application server runtime service that enables you to precisely manage an application's persistence.

The access intent service defines a set of declarative annotations used by the Enterprise JavaBeans (EJB) container and its agents to make performance optimizations for entity bean access. These annotations are organized into sets called *access intent policies*.

Access intent policies contain a set of annotations considered as hints by the EJB container and its agents. Most access intent policies are hints representing high-level abstractions that can be mapped to a specific back end resource manager. It is the responsibility of the EJB persistence machinery to ensure the necessary concurrency control, connection, and cache management when carrying out the persistence details. The EJB persistence manager can use access intent hints to make better performance decisions when carrying out its assigned task. A smaller number of access intents are hints to the EJB container, influencing the management of EJB collections.

Typically, you configure *bean level* access intent for your applications. You can also apply access intent policies to beans within the scope of *application profiles*. Consequently, you can configure beans with multiple and opposing access intent policies. The application profiling documentation explains in more detail how to configure an application to apply a particular access intent policy to a bean for one request, then apply another access intent policy to the same bean for a different request.

Support for applying access intent policies at the method level is deprecated in WebSphere Application Server Version 6.0. In this practice of configuring access intent, you apply a policy to methods within the scope of an EJB module so that the policy becomes the default access intent for all requests upon those methods.

Access intent design considerations

best-practices: Refrain from over-tuning an application. You can introduce errors by incorrectly using the access intent service. For example, misuse of the wsPessimisticUpdate-NoCollision policy can result in lost updates; inappropriately setting the collection increment value can introduce performance issues; and problem determination is more difficult when an application is configured with multiple access intent policies.

Note: Clarity and simplicity should be your guiding principles when using the access intent service. This is even more important when applying access intent polices within the scope of application profiles.

Even though access intent policies can be configured on any method of an entity bean, some attributes of a policy can only be leveraged by the runtime environment under certain conditions. For example, concurrency and access intent are only used for CMP entity beans when the ejbLoad method is driven to open a connection and read data from a given resource; that data is cached and used to drive the proper queries during invocation of the ejbStore method. Read-ahead hints are only used during the execution of a finder for a bean. The collection increment and resource manager prefetch increment are only used on multi-object finders. Configuring policies on methods that do not use the policy is not an error. Only certain attributes of any policy are used, even when the policy is appropriately applied to a method. However, configuring policies unnecessarily throughout an application obscures the design of the application and complicates the maintenance of the application.

Access intent with BMP entity beans

Access intent's declarative functionality provides great power to you as a CMP entity bean developer. You can provide hints on how the product should manage the details of persistence without having to explicitly manage any of the persistence logic in the application. There are situations, however, in which you might need to develop BMP entity beans. Since the only meaningful difference between BMP and CMP components is who provides the persistence logic, BMP entity beans should be able to leverage access intent hints just as the product does on behalf of CMP entity beans. BMP entity beans that use the access intent service participate in application profiling; that is, the value of the access intent attributes can differ from request to request, allowing the BMP entity bean to seamlessly modify its persistence strategy.

You can apply access intent policies to BMP entity bean methods as well as CMP entity bean methods. Because access intent hints are not contractual in nature, there is no obligation for a BMP entity bean to exploit them. BMP entity beans are expected to use only those access intent attributes that are important to that particular bean.

The current access intent policy is bound into the java:comp namespace for a particular BMP entity bean. That policy is current only for the duration of the method call during which the access intent policy was retrieved. In a typical scenario, you would cache the access type during invocation of the ejbLoad method so that appropriate actions can be taken during invocation of the ejbStore method.

Access intent best practices

When applying access intent policies to EJB methods, consider the following issues.

- Start by configuring the default access intent policy for an entity. After your application is built and started, you can tune certain access paths in your application using application profiling or method-level access intent.
- Don't mix access types. Avoid using both pessimistic and optimistic policies in the same transaction. For most databases, pessimistic and optimistic policies use different isolation levels. This can result in multiple database connections, which prevents you from taking advantage of the performance benefits possible through connection sharing.
- Take care when applying the **wsPessimisticUpdate-NoCollision** policy. This policy does not ensure data integrity. No database locks are held, so concurrent transactions can overwrite each other's updates. Use this policy only if you can be sure that only one transaction attempts to update persistent store at any time.

For further information on Java Persistence API (JPA) Access intent, see the topic on JPA Access intent.

Applying access intent policies to methods

You apply an access intent policy to a method, or set of methods, in an application's entity beans through the assembly tool.

About this task

Note: Method-level access intent is deprecated in Version 6.0.

Procedure

1. Start an assembly tool.
2. Optional: Open the Java EE perspective to work with Java EE projects. Click **Window > Open Perspective > Other > Java EE**.
3. Optional: Open the Project Explorer view. Click **Window > Show View > Project Explorer**. Another helpful view is the Navigator view (**Window > Show View > Navigator**).
4. Create a new application EAR file or edit an existing one.

   For example, to change attributes of an existing application, use the import wizard to import an EAR file. To start the import wizard:

   a. Select **File > Import > EAR file > Next**
   b. Select the EAR file.
   c. Create a WebSphere Application Server v6.0 type of Server Runtime. Select **New** to open the New Server Runtime Wizard and follow the instructions.
   d. In the *Target server* field, select *WebSphere Application Server v6.0* type of Server Runtime.
   e. Select **Finish**
5. In the Project Explorer view of the J2EE perspective, right-click the **Deployment Descriptor: EJB Module Name** under the EJB module for the bean instance, then select **Open With > Deployment Descriptor Editor**. A property dialog notebook for the EJB project is displayed in the property pane.
6. Select the **Access** tab.
7. On the right side of the **Access Intent for Entities 2.x (Method Level)** panel, select **Add**. The **Add Access Intent** panel displays.
8. Specify the **Name** for your new intent policy.
9. Select the **Access intent name** from the drop-down list.
10. Enter a **Description** to help you remember what this policy does.
11. Optional: Select **Read Ahead Hint**. A single access intent read ahead hint might not refer to the same bean type in more than one relationship. For example, if a **Department** enterprise bean has a relationship *employees* with the **Employee** enterprise bean, and also has a relationship *manager* with the **Employee** enterprise bean, then a read ahead hint cannot specify both *employees* and *manager*.
12. Click **Next**. The next **Add Access Intent** panel displays, with optional attributes.
13. Optional: Decide whether or not to overwrite these optional access intent attributes. Click on those you want to change.
14. Click **Next**. The next **Add Access Intent** panel, with a list of Enterprise Beans, displays.
15. Select one or more Enterprise Beans from the list.

    Note: If you selected **Read Ahead Hint** in an earlier step, you can only select **ONE** bean at this step.
16. Click **Next**. The next **Add Access Intent** panel, with a list of methods, displays.
17. Select the methods you want to use.
18. If you *DID NOT* select **Read Ahead Hint** in an earlier step, click **Finish**. If you *DID* select the Read Ahead Hint option, you can click **Next** to specify your Read Ahead Hint for the specified bean. The next **Add Access Intent** panel, with a list of EJB preload paths, displays.
19. Edit the EJB preload path by selecting relationship roles from the **Relationship roles:** window.
20. Click **Finish**. A new entry is created in the **Access Intent for Entities 2.x (Method Level)** panel

Developing applications that use the Java Persistence API

Developing JPA 2.x applications for a Java EE environment

Containers in the application server can provide many of the necessary functions for the Java Persistence API (JPA) in a Java Enterprise Edition (Java EE) environment. The application server also provides JPA command tools to assist you with developing applications in a Java EE environment.

About this task

Attention: When you use these JPA command tools, run them from the <*profile_root*>/bin directory, rather than from the *app_server_root*/bin directory to make sure that you have the latest version of the commands for your release level.

For this task, you must specify the `com.ibm.ws.jpa.thinclient_8.0.0.jar` stand-alone Java archive (JAR) file in your class path. This stand-alone JAR file is available from the installation images. The location of this file on the server installation image is ${*app_server_root*}/runtimes/ `com.ibm.ws.jpa.thinclient_8.0.0.jar`..

Important: JPA applications require different configuration techniques from applications that use container-managed persistence (CMP) or bean-managed persistence (BMP). They do not follow the typical deployment techniques that are associated with applications that implement CMP or BMP. In JPA applications, you must define a persistence unit and configure the appropriate properties to ensure that the applications can run in a Java EE environment.

The container supports all necessary injections to ensure that applications run in the Java EE environment. For example, the container can inject the @PersistenceUnit and @PersistenceContext for your applications.

Procedure

1. Generate your entities classes. These are Plain Old Java Object (POJO) entities. Depending upon your development model, you might use some or all of the JPA tools:
   - **Top-down mapping:** You start from scratch with the entity definitions and the object-relational mappings, and then you derive the database schemas from that data. If you use this approach, you are most likely concerned with creating the architecture of your object model and then writing your entity classes. These entity classes would eventually drive the creation of your database model. If you are using a top-down mapping of the object model to the relational model, develop the entity classes and then use OpenJPA functionality to generate the database tables that are based on the entity classes. The **`wsmapping`** tool helps with this approach.
   - **Bottom-up mapping:** You start with your data model, which are the database schemas, and then you work upwards to your entity classes. The **`wsreversemapping`** tool helps with this approach.
   - **Meet in the middle mapping:** probably the most common development model. You have a combination of the data model and the object model partially complete. Depending on the goals and requirements, you must negotiate the relationships to resolve any differences. Both the **`wsmapping`** tool and the **`wsreversemapping`** tool help with this approach.

   The JPA solution for the application server provides several tools that help with developing JPA applications. Combining these tools with IBM Rational Application Developer provides a solid development environment for either Java EE or Java SE applications. Rational Application Developer includes GUI tools to insert annotations, a customized `persistence.xml` file editor, a database explorer, and other features. Another alternative is the Eclipse Dali project. More information about Rational Application Developer or the Eclipse Dali plug-in can be found at their respective websites.
2. Compile the entity classes.

Compile the entities as you would any Java class, unless you are using the Criteria API. If you are using the Criteria API, you must also generate the Criteria API metamodel classes by including the following option with the **javac** command:

```
-Aopenjpa.metamodel=true
```

The following are examples of how you use this option.

For Linux, UNIX and z/OS platforms:

```
app_server_root/java/bin/javac
-Aopenjpa.metamodel=true
-classpath app_server_root/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar
mypackage/MyEntity.java
```

For Windows platform:

```
app_server_root\java\bin\javac
-Aopenjpa.metamodel=true
-classpath app_server_root\runtimes\com.ibm.ws.jpa.thinclient_8.0.0.jar
 mypackage\MyEntity.java
```

Using the **javac** command and the J2EE server plug-in:

For Linux, UNIX and z/OS platforms:

```
app_server_root/java/bin/javac
-Aopenjpa.metamodel=true
-classpath app_server_root/plugins/javax.j2ee.persistence.jar
mypackage/MyEntity.java
```

For Windows platform:

```
app_server_root\java\bin\javac
-Aopenjpa.metamodel=true
-classpath app_server_root\plugins\javax.j2ee.persistence.jar
mypackage\MyEntity.java
```

For more information about using Criteria API and AnnotationProcessor6, refer to the Apache OpenJPA User Guide, Chapter 11: JPA Criteria and Chapter 4: Generation of Canonical Metamodel classes for AnnotationProcessor6 options and the information center topic, Criteria API.

3. Enhance the entity classes using the JPA enhancer tool, **wsenhancer**, for the application server. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. The enhancer post-processes the bytecode that is generated by the Java compiler and adds the fields and methods that are necessary to implement the persistence features. Although JPA for the application server and OpenJPA can automatically enhance the entities at run time, you obtain better performance if you can enhance your entities when you build the application. The application does not attempt to enhance entities that are already enhanced.

   For examples of how to use the **wsenhancer** tool, see the topic, wsenhancer command.

4. Optional: If you are not using the development model for bottom-up mapping, generate or update your database tables automatically or by using the **wsmapping** tool.

   - By default, the object-relational mapping does not occur automatically, but you can configure the application server to provide that mapping with the openjpa.jdbc.SynchronizeMappings property. This property can accelerate development by automatically ensuring that the database tables match the object model. To enable automatic mapping, include the following line in the `persistence.xml` file:

     ```
     <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
     ```

     Note: To enable automatic object-relational mapping at run time, all of your persistent classes must be listed in the Java .class file, mapping file, and Java archive (JAR) file elements in XML format.

   - To manually update or generate your database tables, run the JPA mapping tool for the application server from the command line to create the tables in the database. For examples on how to run the **wsmapping** tool, see the topic, wsmapping command.

5. Optional: If you are using DB2 and want to use static Structured Query Language (SQL), run the **wsdbgen** command. In order to use the **wsdbgen** command, IBM Optim pureQuery Run time must be installed. The **wsdbgen** command creates the *persistence_unit_name*.pdqxml file under the same META-INF directory where your persistence.xml file is located. If you have multiple persistence units, the **wsdbgen** command must be run for each persistence unit.

   When multiple pdqxml files are referenced by an application, use the **Merge** utility to combine them into a single pdqxml file. Specify the combined pdqxml file as pureQueryXml property of pdqProperties. Refer to the Merge utility documentation in the IBM Integrated Data Management information center.

   Note: Applications that implement the JPA and are configured to run static SQL can experience various exceptions. These exceptions might occur with the **wsdbgen** command, which you can use to prepare the application, or when the application is running and calls a JPA method. To resolve this problem, complete the following steps:

   a. Install the program temporary fixes (PTF) for the iSeries JDBC Driver V5R4. Install PTF numbers SI32561 and SI32562. You can find the PTFs through the IBM System i® Support: PTF Cover Letters website.
   b. If you use DB2 Universal Database for iSeries V6R1 or V5R3, visit the fix website for the appropriate release.
   c. Install the required level of pureQuery, which is Version 1.3.100 or later. For more information, see the IBM Optim pureQuery Runtime website. Install the latest JCC driver, which is Version 3.52.95 or later, with the fix for APAR PK65069. The latest JCC drivers are part of the IBM DB2 software package.
   d. For DB2 on a z/OS server, install PTF UK39204 for the V8 alternate driver or PTF UK39205 for V9, and install the fix for APAR PK67706.

   For examples on how to run this command, see the topic, **wsdbgen** command.
6. Optional: If you are using application-managed identity, generate an application-managed identity class with the **wsappid** tool. When you use an application-managed identity, one or more of the fields must be an identity field. Use an identity class if your entity has multiple identity fields and at least one of the fields is related to another entity. The application-managed identity tool generates Java code that uses the identity class for any persistent type that implements application-managed identity.

   For examples on how to use the **wsappid** tool, see the topic wsappid command.

Example

The following is an example of a persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

<persistence-unit name="TheWildZooPU" transaction-type="JTA">
        <jta-data-source>jdbc/DataSourceJNDI</jta-data-source>
        <!-- additional Mapping file, in addition to orm.xml>
        <mapping-file>META-INF/JPAorm.xml</mapping-file>

        <class>com.company.bean.jpa.PersistebleObjectImpl</class>
        <class>com.company.bean.jpa.Animal</class>
        <class>com.company.bean.jpa.Dog</class>
        <class>com.company.bean.jpa.Cat</class>

   <properties>
          <property name="openjpa.ConnectionFactoryProperties"
                    value="PrettyPrint=true, PrettyPrintLineLength=72"/>
          <property name="openjpa.jdbc.SynchronizeMappings"
                    value="buildSchema(ForeignKeys=true)"/>
```

```
        </properties>

    </persistence-unit>
</persistence>
```

What to do next

For more information about the commands, classes or other OpenJPA information, refer to the Apache OpenJPA User Guide.

Developing JPA 2.x applications for a Java SE environment

Prepare persistence applications to test outside of the application server container in a Java SE environment.

About this task

Attention: When you use these JPA command tools, run them from the *profile_root*/`bin` directory, rather than from the *app_server_root*/`bin` directory to make sure that you have the latest version of the commands for your release level.

For this task, you must specify the `com.ibm.ws.jpa.thinclient_8.0.0.jar` stand-alone Java archive (JAR) file in your class path. This stand-alone JAR file is available from the installation images. The location of this file on the server installation image is ${*app_server_root*}/`runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar`..

Important: Java Persistence API (JPA) applications require different configuration techniques from applications that use container-managed persistence (CMP) or bean-managed persistence (BMP). They do not follow the typical deployment techniques that are associated with applications that implement CMP or BMP. In JPA applications, you must define a persistence unit and configure the appropriate properties in the `persistence.xml` file to ensure that the applications can run in a Java SE environment.

There are some considerations for running JPA applications in a Java SE environment:

- Resource injection is not available. You must configure these services specifically or programmatically.
- The life cycle of the EntityManagerFactory and EntityManager are managed by the application. Applications control the creation, manipulation, and deletion of these constructs programmatically.

Procedure

1. Generate your entities classes.

   These are Plain Old Java Object (POJO) entities. Depending upon your development model, you might use some or all of the JPA tools:

   Top-down mapping
   You start from scratch with the entity definitions and the object-relational mappings, and then you derive the database schemas from that data. If you use this approach, you are most likely concerned with creating the architecture of your object model and then writing your entity classes. These entity classes would eventually drive the creation of your database model. If you are using a top-down mapping of the object model to the relational model, develop the entity classes and then use OpenJPA functionality to generate the database tables that are based on the entity classes. The **`wsmapping`** tool would help with this approach.

   Bottom-up mapping
   You start with your data model, which are the database schemas, and then you work upwards to your entity classes. The **`wsreversemapping`** tool would help with this approach.

   Meet in the middle mapping
   Probably the most common development model. You have a combination of the data model

and the object model partially complete. Depending on the goals and requirements, you must negotiate the relationships to resolve any differences. Both the **wsmapping** tool and the **wsreversemapping** tool would help with this approach.

The JPA solution for the application server provides several tools that help with developing JPA applications. Combining these tools with IBM Rational Application Developer provides a solid development environment for either Java EE or Java SE applications. Rational Application Developer includes GUI tools to insert annotations, a customized`persistence.xml` file editor, a database explorer, and other features. Another alternative is the Eclipse Dali project. More information about Rational Application Developer or the Eclipse Dali plug-in can be found at their respective websites.

2. Compile the entity classes.

   Compile the entities as you would any Java class, unless you are using the Criteria API. If you are using the Criteria API, you must also generate the Criteria API metamodel classes by including the following option with the **javac** command:

   ```
   -Aopenjpa.metamodel=true
   ```

   The following are examples of how you use this option.

   ```
   app_server_root/java/bin/javac
   -Aopenjpa.metamodel=true
   -classpath app_server_root/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar
   mypackage/MyEntity.java
   ```

   For more information about using Criteria API and AnnotationProcessor6, refer to the Apache OpenJPA User Guide, Chapter 11: JPA Criteria and Chapter 4: Generation of Canonical Metamodel classes for AnnotationProcessor6 options and the information center topic, Criteria API.

3. Enhance the entity classes using the JPA enhancer tool, or specify the Java agent to perform dynamic enhancement at run time.
   - Use the **wsenhancer** tool. The enhancer post-processes the bytecode that is generated by the Java compiler and adds the fields and methods that are necessary to implement the persistence features. For example examples on how to use the **wsenhancer** tool, see the topic, **wsenhancer** command.
   - You can specify the Java agent mechanism to perform the dynamic enhancement at run time. For example, type the following at the command prompt:

```
java -javaagent:${app_client_root}/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar com.xyz.Main
```

   Attention: You can either run the **wsenhancer** tool or specify the **javaagent** command. You do not need to do both.

4. Optional: If you are not using the development model for bottom-up mapping, generate or update your database tables automatically or by using the **wsmapping** tool.
   - By default, the object-relational mapping does not occur automatically, but you can configure the application server to provide that mapping with the openjpa.jdbc.SynchronizeMappings property. This property can accelerate development by automatically ensuring that the database tables match the object model. To enable automatic mapping, include the following line in the `persistence.xml` file:

     ```
     <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)"/>
     ```

     Note: To enable automatic object-relational mapping at run time, all of your persistent classes must be listed in the Java `.class` file, mapping file, and Java archive (JAR) file elements in XML format.
   - To manually update or generate your database tables, run the JPA mapping tool for the application server from the command line to create the tables in the database. For examples on how to run the **wsmapping** tool, see the topic, wsmapping command.
5. Optional: If you are using DB2 and want to use static Structured Query Language (SQL), run the **wsdbgen** command. In order to use the **wsdbgen** command, IBM Optim™ PureQuery Run time must be installed. The **wsdbgen** command creates the *persistence_unit_name*`.pdqxml` file under the same META-INF directory where your `persistence.xml` file is located. If you have multiple persistence units, the **wsdbgen** command must be run for each persistence unit.

When multiple pdqxml files are referenced by an application, use the **Merge** utility to combine them into a single pdqxml file. Specify the combined pdqxml file as pureQueryXml property of pdqProperties. See the Merge utility documentation in the IBM Integrated Data Management information center.

Note: Applications that utilize JPA and are configured to run static SQL can experience various exceptions. These exceptions might occur with the **wsdbgen** command, which you can use to prepare the application, or when the application is running and calls a JPA method. To resolve this problem, complete the following:

a. Install the program temporary fixes (PTF) for the iSeries JDBC Driver V5R4. Install PTF numbers SI32561 and SI32562. You can find the PTFs through the IBM System i Support: PTF Cover Letters website.

b. If you use DB2 Universal Database for iSeries V6R1 or V5R3, visit the fix web site for the appropriate release.

c. Install the required level of IBM Optim PureQuery Run time, which is Version 1.3.100 or later. For more information, see the IBM Data Studio pureQuery Runtime website. Install the latest JCC drivers, which is Version 3.52.95 or later, with the fix for APAR PK65069. The latest JCC drivers are part of the IBM DB2 software package.

d. For DB2 on a z/OS server, install PTF UK39204 for the V8 alternate driver or PTF UK39205 for V9, and install the fix for APAR PK67706.

For examples on how to run this command, see the topic, **wsdbgen** command.

6. Optional: If you are using application-managed identity, generate an application-managed identity class with the **wsappid** tool. When you use an application-managed identity, one or more of the fields must be an identity field. Use an identity class if your entity has multiple identity fields and at least one of the fields is related to another entity. The application-managed identity tool generates Java code that uses the identity class for any persistent type that implements application-managed identity.

   For examples on how to use the **wsappid** tool, see the topic wsappid command.

Example

The following is a sample persistence.xml file for the Java SE Environment:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="TheWildZooPU" transaction-type="RESOURCE_LOCAL">

        <!-- additional Mapping file, in addition to orm.xml>
        <mapping-file>META-INF/JPAorm.xml</mapping-file>

        <class>com.company.bean.jpa.PersistebleObjectImpl</class>
        <class>com.company.bean.jpa.Animal</class>
        <class>com.company.bean.jpa.Dog</class>
        <class>com.company.bean.jpa.Cat</class>



        <properties>
            <property name="openjpa.ConnectionDriverName"
                      value="org.apache.derby.jdbc.EmbeddedDriver" />
            <property name="openjpa.ConnectionURL"
                      value="jdbc:derby:target/database/jpa-test-database;create=true" />
            <property name="openjpa.Log"
                      value="DefaultLevel=INFO,SQL=TRACE,File=./dist/jpaEnhancerLog.log,Runtime=INFO,Tool=INFO" />
            <property name="openjpa.ConnectionFactoryProperties"
                      value="PrettyPrint=true, PrettyPrintLineLength=72" />
            <property name="openjpa.jdbc.SynchronizeMappings"
                      value="buildSchema(ForeignKeys=true)" />
            <property name="openjpa.ConnectionUserName"
                      value="user" />
```

```
            <property name="openjpa.ConnectionPassword"
                      value="password"/>
        </properties>

    </persistence-unit>

</persistence>
```

What to do next

For more information about any of the commands, classes or other OpenJPA information, refer to the Apache OpenJPA User Guide.

Bean validation in JPA

Data validation is a common task that occurs in all layers of an application, including persistence. The Java Persistence API (JPA) 2.0 provides support for the Bean Validation API so that data validation can be done at run time. This topic includes a usage scenario where bean validation is used in the JPA environment of a sample digital image gallery application.

The Bean Validation API provides seamless validation across technologies on Java Enterprise Edition 6 (Java EE 6) and Java Platform, Standard Edition (JSE) environments. In addition to JPA 2.0, these technologies include JavaServer Faces (JSF) 2.0 and Java EE Connector Architecture (JCA) 1.6. You can read more about bean validation in the topic, Bean Validation API.

There are three core concepts of bean validation: constraints, constraint violation handling, and the validator. If you are running applications in an integrated environment like JPA, there is no need to interface directly with the validator.

Validation constraints are annotations or XML code that are added to a class, field, or method of a JavaBeans component. Constraints can be built in or user-defined. They are used to define regular constraint definitions and for composing constraints. The built-in constraints are defined by the bean validation specification and are available with every validation provider. For a list of built-in constraints, see the topic, Bean validation built-in constraints. If you need a constraint different from the built-in constraints, you can build your own user-defined constraint.

Constraints and JPA

The following usage scenario illustrates how a built-in constraint is used in the JPA architecture of a sample digital image gallery application.

In the first code example, a built-in constraint is added to a simple entity of the JPA model called *image*. An image has an ID, image type, file name, and image data. The image type must be specified and the image file name must include a valid JPEG or GIF extension. The code shows the annotated image entity with some built-in bean validation constraints applied.

```
package org.apache.openjpa.example.gallery.model;

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;

@Entity
public class Image {

    private long id;
    private ImageType type;
```

```
    private String fileName;
    private byte[] data;

    @Id
    @GeneratedValue
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    @NotNull(message="Image type must be specified.")
    @Enumerated(EnumType.STRING)
    public ImageType getType() {
        return type;
    }

    public void setType(ImageType type) {
        this.type = type;
    }

    @Pattern(regexp = ".*\\.jpg|.*\\.jpeg|.*\\.gif",
        message="Only images of type JPEG or GIF are supported.")
    public String getFileName() {
        return fileName;
    }

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public byte[] getData() {
        return data;
    }

    public void setData(byte[] data) {
        this.data = data;
    }
}
```

The Image class uses two built-in constraints, @NotNull and @Pattern. The @NotNull constraint ensures that an ImageType element is specified and the @Pattern constraint uses regular expression pattern matching to ensure that the image file name is suffixed with a supported image format. Each constraint has corresponding validation logic that gets started at run time when the image entity is validated. If either constraint is not met, the JPA provider throws a ConstraintViolationException with the defined message. The JSR-303 specification also makes provisions for the use of a variable within the message attribute. The variable references a keyed message in a resource bundle. The resource bundle supports environment-specific messages and globalization, translation, and multicultural support of messages.

You can create your own custom validator and constraints. In the previous example, the Image entity used the @Pattern constraint to validate the file name of the image. However, it did not check constraints on the actual image data itself. You can use a pattern-based constraint; however, you do not have the flexibility that you would if you created a constraint specifically for checking constraints on the data. In this case you can build a custom method-level constraint annotation. The following is a custom or user-defined constraint called ImageContent.

```
package org.apache.openjpa.example.gallery.constraint;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
```

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import javax.validation.Constraint;
import javax.validation.Payload;

import org.apache.openjpa.example.gallery.model.ImageType;

@Documented
@Constraint(validatedBy = ImageContentValidator.class)
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
public @interface ImageContent {
     String message() default "Image data is not a supported format.";
     Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
  ImageType[] value() default { ImageType.GIF, ImageType.JPEG };
}
```

Next, you must create the validator class, ImageContentValidator. The logic within this validator gets implemented by the validation provider when the constraint is validated. The validator class is bound to the constraint annotation through the validatedBy attribute on the @Constraint annotation as shown in the following code:

```
package org.apache.openjpa.example.gallery.constraint;
import java.util.Arrays;
import java.util.List;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.apache.openjpa.example.gallery.model.ImageType;
/**
 * Simple check that file format is of a supported type
 */
public class ImageContentValidator implements ConstraintValidator<ImageContent, byte[]> {
    private List<ImageType> allowedTypes = null;
    /**
     * Configure the constraint validator based on the image
     * types it should support.
     * @param constraint the constraint definition
     */
    public void initialize(ImageContent constraint) {
        allowedTypes = Arrays.asList(constraint.value());
    }
    /**
     *Validate a specified value.
     */
    public boolean isValid(byte[] value, ConstraintValidatorContext context) {
        if (value == null) {
            return false;
        }
        // Verify the GIF header is either GIF87 or GIF89
        if (allowedTypes.contains(ImageType.GIF)) {
            String gifHeader = new String(value, 0, 6);
            if (value.length >= 6 &&
                (gifHeader.equalsIgnoreCase("GIF87a") ||
                 gifHeader.equalsIgnoreCase("GIF89a"))) {
                return true;
            }
        }
        // Verify the JPEG begins with SOI and ends with EOI
        if (allowedTypes.contains(ImageType.JPEG)) {
            if (value.length >= 4 &&
                value[0] == 0xff && value[1] == 0xd8 &&
                value[value.length - 2] == 0xff && value[value.length -1] == 0xd9) {
                return true;
            }
        }
```

```
        // Unknown file format
        return false;
    }
}
```

Apply this new constraint to the getData() method on the Image class; for example:

```
@ImageContent
    public byte[] getData() {
        return data;
    }
```

When validation of the data attribute occurs, the isValid() method in the ImageContentValidator is started. This method contains logic for performing simple validation of the format of the binary image data. A potentially overlooked feature in the ImageContentValidator is that it can also validate for a specific image type. By definition, it accepts JPEG or GIF formats, but it can also validate for a specific format. For example, by changing the annotation to the following code example, the validator is instructed to only permit image data with valid JPEG content:

```
@ImageContent(ImageType.JPEG)
    public byte[] getData() {
        return data;
    }
```

Type-level constraints are also a consideration because you might need to validate combinations of attributes on an entity. In the previous examples validation constraints were used on individual attributes. Type-level constraints make it possible to provide collective validation. For example, the constraints applied to the image entity validate that an image type is set (not null), the extension on the image file name is of a supported type, and the data format is correct for the indicated type. But, for example, it does not collectively validate that a file named `img0.gif` is of type GIF and the format of the data is for a valid GIF file image. For more information about type-level constraints, see the white paper, OpenJPA Bean Validation Primer, and the section "Type-level constraints."

Validation groups

Bean validation uses validation groups to determine what type of validation and when validation occurs.

There are no special interfaces to implement or annotations to apply to create a validation group. A validation group is denoted by a class definition.

Note: When using groups, use simple interfaces. Using a simple interface makes validation groups more usable in multiple environments. Whereas, if a class or entity definition is used as a validation group, it might pollute the object model of another application by bringing in domain classes and logic that do no make sense for the application. By default, if a validation group or multiple groups is not specified on an individual constraint, it is validated using the javax.validation.groups.Default group. Creating a custom group is as simple as creating a new interface definition.

For more information about validation groups, read the white paper, OpenJPA Bean Validation Primer, and the section "Validation groups."

JPA domain model

In addition to the Image entity are Album, Creator and Location persistent types. An Album entity contains a reference to collection of its Image entities. The Creator entity contains a reference to the album entities that the image Creator contributed to and a reference to the Image entities created. This provides full navigational capabilities to and from each of the entities in the domain. An embeddable location, has been added to image to support storing location information with the image.

The Album and Creator entities have standard built-in constraints. The embeddable location is more unique in that it demonstrates the use of the @Valid annotation to validate embedded objects. To embed location into an image, a new field and corresponding persistent properties are added to the Image class; for example:

```
private Location location;

    @Valid
    @Embedded
    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }
```

The @Valid annotation provides chained validation of embeddable objects within a JPA environment. Therefore, when image is validated, any constraints on the location it references are also validated. If @Valid is not specified, the location is not validated. In a JPA environment, chained validation through @Valid is only available for embeddable objects. Referenced entities and collections of entities are validated separately to prevent circular validation.

Bean validation and the JPA environment

The JPA 2.0 specification makes integration with the Bean Validation API simple. In a JSE environment, bean validation is enabled by default when you provide the Bean Validation API and a bean validation provider on your runtime class path. In a Java EE 6 environment, the application server includes a bean validation provider so there is no need to bundle one with your application. In both environments, you must use a Version 2.0 `persistence.xml` file.

A Version 1.0 `persistence.xml` provides no means to configure bean validation. Requiring a Version 2.0 `persistence.xml` prevents a pure JPA 1.0 application from incurring the validation startup and runtime costs. This is important given that there is no standard means for a 1.0-based application to disable validation. In a Java EE 6 environment, enable validation in an existing 1.0 application by modifying the root element of your `persistence.xml` file. The following example represents the `persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0" >
...
</persistence>
```

Bean validation provides three modes of operation within the JPA environment:

- Auto

  Enables bean validation if a validation provider is available within the class path. Auto is the default.
- Callback

  When callback mode is specified, a bean validation provider must be available for use by the JPA provider. If not, the JPA provider throws an exception upon instantiation of a new JPA entity manager factory.
- None

  Disables bean validation for a particular persistence unit.

Auto mode simplifies deployment, but can lead to problems if validation does not take place because of a configuration problem.

Note: Use either none or callback mode explicitly for consistent behavior.
Also, if none is specified, JPA optimizes at startup and does not attempt to perform unexpected validation. Explicitly disabling validation is especially important in a Java EE 6 environment where the container is mandated to provide a validation provider. Therefore, unless specified, a JPA 2.0 application started in a container has validation enabled. This process adds additional processing during life cycle events.

There are two ways to configure validation modes in JPA 2.0. The simplest way is to add a validation-mode element to the `persistence.xml` with the wanted validation mode as shown in the following example:

```
 <persistence-unit name="auto-validation">
          ...
    <!-- Validation modes: AUTO, CALLBACK, NONE -->
    <validation-mode>AUTO</validation-mode>
    ...
  </persistence-unit>
```

The other way is to configure the validation mode programmatically by specifying the javax.persistence.validation.mode property with value auto, callback, or none when creating a new JPA entity manager factory as shown in the following example:

```
Map<String, String> props = new HashMap<String, String>();
  props.put("javax.persistence.validation.mode", "callback");
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("validation", props);
```

Bean validation within JPA occurs during JPA life cycle event processing. If enabled, validation occurs at the final stage of the PrePersist, PreUpdate, and PreRemove life cycle events. Validation occurs only after all user-defined life cycle events, since some of those events can modify the entity that is being validated. By default, JPA enables validation for the default validation group for PrePersist and PreUpdate life cycle events. If you must validate other validation groups or enable validation for the PreRemove event, you can specify the validation groups to validate each life cycle event in the `persistence.xml` as shown in the following example:

```
<persistence-unit name="non-default-validation-groups">
  <class>my.Entity</class>
  <validation-mode>CALLBACK</validation-mode>
  <properties>
   <property name="javax.persistence.validation.group.pre-persist"
       value="org.apache.openjpa.example.gallery.constraint.SequencedImageGroup"/>
   <property name="javax.persistence.validation.group.pre-update"
       value="org.apache.openjpa.example.gallery.constraint.SequencedImageGroup"/>
   <property name="javax.persistence.validation.group.pre-remove"
       value="javax.validation.groups.Default"/>
  </property>
 </persistence-unit>
```

The following example shows various stages of the JPA life cycle, including persist, update, and remove:

```
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("BeanValidation");
        EntityManager em = emf.createEntityManager();

        Location loc = new Location();
        loc.setCity("Rochester");
        loc.setState("MN");
        loc.setZipCode("55901");
        loc.setCountry("USA");

        // Create an Image with non-matching type and file extension
        Image img = new Image();
        img.setType(ImageType.JPEG);
        img.setFileName("Winter_01.gif");
        loadImage(img);
        img.setLocation(loc);
```

```
// *** PERSIST ***
try {
    em.getTransaction().begin();
    // Persist the entity with non-matching extension and type
    em.persist(img);
} catch (ConstraintViolationException cve) {
    // Transaction was marked for rollback, roll it back and
    // start a new one
    em.getTransaction().rollback();
    em.getTransaction().begin();
    // Fix the file type and re-try the persist.
    img.setType(ImageType.GIF);
    em.persist(img);
    em.getTransaction().commit();
}

// *** UPDATE ***
try {
    em.getTransaction().begin();
    // Modify the file name to a non-matching file name
    // and commit to trigger an update
    img.setFileName("Winter_01.jpg");
    em.getTransaction().commit();
}  catch (ConstraintViolationException cve) {
    // Handle the exception.  The commit failed so the transaction
    // was already rolled back.
    handleConstraintViolation(cve);
}
// The update failure caused img to be detached. It must be merged back
// into the persistence context.
img = em.merge(img);

// *** REMOVE ***
em.getTransaction().begin();
try {
    // Remove the type and commit to trigger removal
    img.setType(ImageType.GIF);
    em.remove(img);
}  catch (ConstraintViolationException cve) {
    // Rollback the active transaction and handle the exception
    em.getTransaction().rollback();
    handleConstraintViolation(cve);
}
em.close();
emf.close();
```

Exceptions

Validation errors can occur in any part of JPA life cycle.

If one or more constraints fail to validate during a life cycle event, a ConstraintViolationException is thrown by the JPA provider. The ConstraintViolationException thrown by the JPA provider includes a set of ConstraintViolations that occurred. Individual constraint violations contain information regarding the constraint, including: a message, the root bean or JPA entity, the leaf bean which is useful when validating JPA embeddable objects, the attribute which failed to validate, and the value that caused the failure. The following is a sample exception handling routine:

```
private void handleConstraintViolation(ConstraintViolationException cve) {
      Set<ConstraintViolation<?>> cvs = cve.getConstraintViolations();
   for (ConstraintViolation<?> cv : cvs) {
     System.out.println("------------------------------------------------");
          System.out.println("Violation: " + cv.getMessage());
          System.out.println("Entity: " + cv.getRootBeanClass().getSimpleName());
          // The violation occurred on a leaf bean (embeddable)
          if (cv.getLeafBean() != null && cv.getRootBean() != cv.getLeafBean()) {
```

```
            System.out.println("Embeddable: " +
cv.getLeafBean().getClass().getSimpleName());
          }
          System.out.println("Attribute: " + cv.getPropertyPath());
          System.out.println("Invalid value: " + cv.getInvalidValue());
      }
    }
```

Constraint violation processing is typically simple when using attribute-level constraints. If you are using a type-level validator with type-level constraints, it can be more difficult to determine which attribute or combination of attributes failed to validate. Also, the entire object is returned as the invalid value instead of an individual attribute. In cases where specific failure information is required, use of an attribute-level constraint or a custom constraint violation might be provided as described in the Bean Validation specification.

Sample

The JPA model and image gallery application usage scenario provided in this topic can be implemented through a sample that is provided in the white paper, *OpenJPA Bean Validation primer*.

wsjpa properties

The extension properties of Java Persistence API (JPA) for WebSphere Application Server can be specified with the openjpa or wsjpa prefix. This topic features the wsjpa properties.

wsjpa.AccessIntent

Use this property to define a TaskName that in the `persistence.xml` file using the wsjpa.AccessIntent property name in a persistence unit. The property value is a list of TaskNames, entity types and access intent definitions.

For more information and examples on how the wsjpa.AccessIntent property is used, see the topic Specifying TaskName in a JPA persistence unit.

wsjpa.jdbc.Schema

Specifies the schema name in a DB2 package collection when using multiple DB2 package collections.

For more information about using the wsjpa.jdbc.Schema property see the topic, Configuring pureQuery to use multiple DB2 package collections.

wsjpa.jdbc.CollectionId

Specifies the collection Id name in a DB2 package collection when using multiple DB2 package collections.

For more information about using the wsjpa.jdbc.CollectionId property see the topics, Configuring pureQuery to use multiple DB2 package collections and Configuring data source JDBC providers to use pureQuery in a Java SE environment.

Criteria API

The Criteria API is an API for building queries with Java objects, as an alternative to building strings for Java Persistence Query Language (JPQL) queries.

The Criteria API supports building queries dynamically at run time, and also the ability to build type-safe queries that can be verified by the compiler. The correctness of JPQL queries cannot be verified by the compiler, and must be verified at run time during testing.

The following is a sample JPQL query that returns a list of employees with less than five years of service:

```
SELECT e FROM Employee e WHERE e.serviceyears < 5
```

Here is a sample of the equivalent Criteria query:

```
QueryBuilder qb = emf.getQueryBuilder();
CriteriaQuery q = qb.create(Employee.class);
Root e = q.from(Employee.class);
q.where(qb.lt(e.get(Employee_.serviceyears), 5));
TypedQuery tq = em.createQuery(q);
List result = q.getResultList();
```

Note: Employee_ is the Metamodel of the Employee class.

Two important features are improvements from JPQL:

- The Criteria API can express queries that are not possible through JPQL. For more detailed information see the section, “Editable data store expressions”, in the developerWorks article, Dynamic, typesafe queries in JPA 2.0.
- A CriteriaQuery can be edited programmatically. For more information see the section, “Editable query”, in the developerWorks article, Dynamic, typesafe queries in JPA 2.0.

You can read more about the Criteria API in the Apache OpenJPA User Guide.

wsappid command

The Java Persistence API (JPA) specification supports an entity primary key to be made up of more than one column. In this case, the primary key is called a composite or compound primary key. Provide an ID class, which is specified by the @IdClass annotation, to manage a composite primary key. Use the identity tool for JPA to generate an ID class for entities that use composite primary keys.

Syntax

Before running the command, you must have a copy of the `persistence.xml` file on the classpath, or specify it as a properties file in the -p [*path_to_persistence.xml*] argument. Issue the command from the `bin` subdirectory of the *profile_root* directory.

The command syntax is as follows:

```
wsappid.sh [parameters][arguments]
```

Parameters

The wsappid tool accepts the standard set of command-line arguments that are defined by the configuration framework along with the following:

- **`-directory/-d`** *<output_directory>*: The path to the output directory.

  If the directory does not match the generated output ID class package, the package structure is created beneath the directory. If this parameter is not specified, the wsappid tool attempts to find the directory of the `.java` file for the class that supports persistence. If a `.java` file is not found, the wsappid tool uses the current directory.
- **`-ignoreErrors/-i`** `<true/t | false/f>`: If this parameter is set to `false`, an exception is occurs when the tool is run on any class that does not use the application identity. An error also occurs if any class does not have the base class in the inheritance hierarchy.
- **`-token/-t`** *<token>*: The token that is used to separate the values of strung primary keys in the string form of the object ID.

  Use this option only if there are multiple primary key fields. The default is "`::`".
- **`-name/-n`** *<id_class_name>*: The name of the identity class to generate.

  If this option is specified, the wsappid tool must run on exactly one class. If the class metadata already names an ID class for the object, this option is ignored. If the name is not fully qualified, the package of the persistence class is appended to form the fully qualified name.
- **`suffix`** *<id_class_suffix>*: A string with which to suffix each persistent class name to form the identity class name.

This option is overridden by the **-name/-n** parameter or by any object ID class that is specified in the metadata.

Each additional argument to the wsappid tool must be one of the following:
- The full name of a persistent class.
- The `.java` name for a persistent class.
- The `.class` file of a persistent class.

Usage

The identity tool used with JPA for application server simplifies the task of creating an identity class for entities that use composite IDs. A composite ID is an identity with more than one field as its primary key. The entity class must be compiled, and primary keys must be identified in the entity class. Run the wsappid tool from the command line in the *profile_root*/bin/ directory. When you run this command, a new class representing the composite ID of the entity is generated. Messages and errors are logged to the console as specified.

Examples

Consider the following entity:

```
@Entity
public class Employee {

@Id
private int division;

@Id private int id;
// . . .
}
```

Before the entity is used, an ID class is needed. For this example, assume that the entity is found in the `src/main/java` directory.

To generate an ID class for the Magazine entity run the following:

```
wsappid.sh -s Id src/main/java/Employee.java -d src/main/java
```

A new class, EmployeeId.java, is generated in the `src/main/java` directory.

Additional information

Read the Application identity tool in persistence classes information in the Apache OpenJPA User Guide for more information.

wsenhancer command

The entity enhancer tool for Java Persistence API (JPA) applications inserts bytecode into an entity class file that supports the JPA provider to manage the state of an entity.

JPA with the application server requires that all entity classes be enhanced if you want to manage their state. In a container-managed environment, automated enhancement is provided by the containers. In a Java SE environment, though, there are no containers to manage persistence and you might use this command frequently before packaging application files for testing. After you have created the JPA entities, you can run the wsenhancer tool to inject bytecode into the entities before packaging the Java archive (JAR) file into the enterprise archive (EAR) file for the application.

Syntax

Before running the command, you must have a copy of the `persistence.xml` file on the classpath, or specify it as a properties file in the -p [*path_to_persistence.xml*] argument. Issue the command from the `bin` subdirectory of the *profile_root* directory.

The command syntax is as follows:

```
wsenhancer.sh [parameters][arguments]
```

Parameters

The enhancer accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-directory/-d** *<output directory>*: Specifies the path to the output directory.

  If the directory does not match the enhanced class package, the package structure is created beneath the directory. By default, the enhancer overwrites the original `.class` file.
- **-enforcePropertyRestrictions/-epr** `<true/t | false/f>`: Specifies whether to generate an exception when a property access entity is not obeying the restrictions that are placed on property access.

  The default is set to `false`.
- **-addDefaultConstructor/-adc** `<true/t | false/f>`: Specifies that all of the persistent classes define a no-argument constructor. This flag informs the enhancer to add a protected no-arg constructor to any persistent classes in which the constructor is not already present.
- **-tmpClassLoader/-tcl** `<true/t | false/f>`: Specifies whether the enhancer should load persistent classes with a temporary class loader.

  This function supports other code to load the enhanced version of the class afterward within the same Java virtual machine (JVM). The default is set to true.

  Note: If you are encountering class loading problems when running the enhancer, you can set this flag to `false` as a debugging step.
- For class name, specify one of the following:
  - The full name of a class.
  - The `.java` name for a class.
  - The `.class` file of a class.

If you do not provide arguments to the enhancer, it runs on the classes in your persistent class list.

Usage

To use the wsenhancer tool you need entities defined to the JPA specifications, and the entities must be compiled. Run the wsenhancer tool against the entities before packaging them into a JAR file. If the entities are already packaged, you extract the entity class files, run the enhancer, and recreate the JAR file.

To enhance your entities:

- Verify that your entities are in the class path, if they are not, add them.
- Run the wsenhancer command. It is found in ${profile_root}/bin directory.

Messages and errors are logged to the administrative console as specified in the log settings. After starting the wsenhancer command, your files are enhanced.

Examples

To enhance all entities on the class path:

```
$ cd build
/home/user/myproject/build $ ${profile_root}/bin/wsenhancer.sh
```

All entities in myproject are enhanced.

To enhance a specific entity when you have the source files:

```
$ cd build
/home/user/myproject/build $ ${profile_root}/bin/wsenhancer.sh Magazine.java
```

To enhance a specific entity when you have the compiled class files:

```
$ export CLASSPATH=target/classes
$ ${profile_root}/bin/wsenhancer.sh /bin/wsenhancer.sh target/classes/jpa/example/MyEntity.class
```

The entity, Magazine.java, located in project are enhanced.

Additional information

For more information about enhancement tools, see the section on persistent classes in the Apache OpenJPA documentation.

wsmapping command

The wsmapping tool is used to provide top-down mapping of the entity object model to the database relational model. You can use the wsmapping tool to create database tables.

Syntax

Before running the command, you must have a copy of `persistence.xml` on the class path, or specify it as a properties file in the -p [*path_to_persistence.xml*] argument. Issue the command from the `bin` subdirectory of the *profile_root* directory.

The command syntax is as follows:

```
wsmapping.sh [options][arguments]
```

Parameters

The mapping tool accepts the standard set of command-line arguments defined by the configuration framework with the following options:

- **-schemaAction/-sa** <`add` | `refresh` | `drop` | `build` | `reflect` | `retain` | `createDB` | `import` | `export` | `none`>: The action to implement against the schema.

  These options correspond to the actions of the schema tool. `Add` is the default action if none is specified. Actions can be composed in a list separated by commas.

  Note: The wsmapping tool accepts the -action/-a flag to specify the action to take on individual classes. Unless you are running wsmapping on all of your persistent types at once, or dropping a mapping, you must use the default `add` action or the `build` action. Otherwise, you might inadvertently drop schema components that are used by classes that you are not currently running the tool against.
- **-schemaFile/-sf** <`true/t` | `false/f`>: This option can be used to write the planned schema to an XML document rather than modify the database.

  The XML document can then be modified, manipulated and committed to the database with the schema tool.
- **-sqlFile/-sql** <`stdout` | `output file`>: This option can be used to write the planned schema modifications to an SQL script rather than modify the database.

  Combine this parameter with a **schemaAction** of `build` to generate a script that recreates the schema for the current mappings, even if the schema exists.
- **-dropTables/-dt** <`true/t` | `false/f`>: When this option is set to true, schema drops tables that appear to be unused during `retain` and `refresh` actions.

The default is `true`.

- **`-dropSequences/-dsq`** <`true/t` | `false/f`>: If this option is set to `true`, schema drops sequences that are unused during `retain` and `refresh` actions.

  The default is `true`.
- **`-openjpatables/-ot`** <`true/t` | `false/f`>: When reflecting the schema, this parameter determines whether to reflect on tables and sequences with names that start with OPENJPA_.

  Certain OpenJPA components use these tables and sequences, such as the table schema factory. When using other actions, the openjpaTables parameter controls whether these tables can be dropped or not. The default setting is `false`.
- **`-ignoreErrors/-i`** <`true/t` | `false/f`>: If set to `false`, an exception is occurs if the tool encounters database errors.

  The default is set to `false`.
- **`-schemas/-s`** *<schema list>*: Denotes a list of schema and table names the OpenJPA should access when running the wsschema tool.

  This is the equivalent to setting the openjpa.jdbc.Schemas property to run once. This parameter corresponds to the **`-schemas/-s`** parameter in the wsschema tool. This option is ignored if **`-readSchema/-rs`** is not set to `true`.
- **`-readSchema/-rs`** <`true/t` | `false/f`>: Set this option to `true` to read the entire existing schema when the mapping tool runs.

  Reading the existing schema ensures that OpenJPA does not generate any mappings that use the table, index, primary key or foreign key names that conflict with existing names.

  Note: Depending on the particular JDBC driver, selecting the **`-readSchema/-rs`** function can slow down the process for large schemas.
- **`-primaryKeys/-pk`** <`true/t` | `false/f`>: This flag determines if the primary keys can be manipulated on existing tables.

  The default is `true`.
- **`-foreignKeys/fk`** <`true/t` | `false/f`>: This flag determines if foreign keys can be manipulated on existing tables.

  The default is `true`. This means that to add a new foreign key to a class that has already been mapped, you must explicitly set this parameter flag to `true`.
- **`-indexes/-ix`** <`true/t` | `false/f`>: This flag determines if indexes can be manipulated on existing tables.

  The default is `true`. This means that to add new indexes to a class that has already been mapped, you must explicitly set this parameter flag to `true`.
- **`-sequences/-sq`** <`true/t` | `false/f`>: This flag determines if sequences can be manipulated.

  The default is `true`.
- **`-meta/-m`** <`true/t` | `false/f`>: This flag determines whether a mapping applies to metadata rather than, or in addition to, standard mappings.
- The wsmapping tool accepts the **`-action/-a`** flag to specify the action to take on each class. Multiple actions can be composed in a list, separated by commas. The available actions are:
  - **`buildSchema`**: This is the default action. The **`buildSchema`** action makes the database schema match your existing mappings. If the provided mappings conflict with the class definitions, OpenJPA fails with an informative exception.
  - **`validate`**: Ensure that the mappings for the given classes are valid and that they match the schema of the database. No mappings of tables are changed as a result of this action. An exception occurs if any mappings are invalid.

Each additional argument to the wsmapping tool must be one of the following:

- The full name of a persistent class.
- The `.java` name for a persistent class.
- The `.class` file of a persistent class.

If you do not supply any arguments to the wsmapping tool, it runs on the classes in the persistent classes list.

Usage

Before running the wsmapping tool, you must configure the data source information, including the URL, user, and password. It is required that the wsenhancer tool is run before the wsmapping tool to insert bytecode into the entity classes. Also, the compiled class files for your entities should be on the class path. Assume that entity class files can be found in target/classes, for example:

```
export CLASSPATH=${CLASSPATH}:target/classes

wsmapping.sh ...
```

To create tables, run the wsmapping command from the ${profile_root}/bin directory. When completed, the database tables are created or updated. Messages and errors are logged to the administrative console as specified by log settings.

wsmapping.sh . . . On Windows :

Tip: By specifying the buildSchema parameter to the openjpa.jdbc.SynchronizeMappings property, the mapping tool provides the default mapping that matches with the database schema automatically. You are not required to run this mapping tool if the default mapping satisfies the necessary database schema.

Examples

To create the database tables needed for the `Magazine.java` file:

```
${profile_root}/bin/wsmapping.sh Magazine.java
```

To drop the tables for `Magazine.java`:

```
C:\> %profile_root%/bin/wsmapping.sh -sa dropDB Magazine.java
```

To validate the mappings for all classes on the class path:

```
C:\> %profile_root%/bin/wsmapping.sh -a validate
```

Additional information

See the mapping information in the Apache OpenJPA User Guide for more information and examples.

wsreversemapping command

The wsreversemapping tool generates persistent class definitions and metadata from a database schema.

Syntax

Before running the command, you must have a copy of the `persistence.xml` file on the class path, or specify it as a properties file in the -p [*path_to_persistence.xml*] argument. Issue the command from the `bin` subdirectory of the *profile_root* directory.

The command syntax is as follows:

```
wsreversemapping.sh [parameters][arguments]
```

Parameters

The wsreversemapping tool accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-schemas/-s** *<schema and table names>*: A list of schema and table names, separated by commas, to run the wsreversmapping tool on if no XML schema file is supplied.

  Each element of the list must follow the naming conventions for the openjpa.jdbc.Schemas property. If this parameter flag is omitted, it defaults to the value of the **Schemas** property. If the **Schemas** property is not defined, then all schemas are reverse mapped.
- **-package/-p** *<package name>*: The package name of the generated classes.

  If no package name is given, the generated code does not contain package declarations.
- **-directory/-d** *<output directory>*: All generated code and metadata is written to the directory at this path.

  If the path does not match the package of a class, the package structure is created beneath this directory. This parameter defaults to the current directory.
- **-useSchemaName/-sn** <true/t | false/f>: Set this parameter flag to true to include the schema and the table name in the name of each generated class.

  This method can be useful when dealing with multiple schemas that have tables with identical names.
- **-useForeignKeyName/-fkn** <true/t | false/f>: Set this parameter flag to true if you want the field names for relations to be based on the database foreign key name.

  By default, relation field names are derived from the name of the related class.
- **-nullableAsObject/-no** <true/t | false/f>: By default, all non-foreign key columns are mapped to primitives.

  Set this parameter flag to true to generate primitive wrapper fields instead for columns that support null values.
- **-blobAsObject/-bo** <true/t | false/f>: By default, all binary columns are mapped to the byte[] fields.

  Set this parameter flag to true to map them to Object fields instead.

  Attention: When mapped this way, the column is presumed to contain a serialized Java object.
- **-primaryKeyOnJoin/pkj** <true/t | false/f>: The standard reverse mapping tool behavior is to map all tables with primary keys to persistent classes.

  If your schema has primary keys on many join tables as well, set this flag to true to avoid creating classes for those tables.
- **-inverseRelations/-ir** <true/t | false/f>: Set this parameter flag to false to prevent the creation of inverse one-to-many or one-to-one relations for every many-to-one or one-to-one relation detected.
- **-useDatastoreIdentity/-ds** <true/t | false/f>: Set to true to use data store identity for tables that have single numeric primary key columns.

  Typically, the tool uses application identity for all generated classes.
- **-useBuiltinIdentityClass/-bic** <true/t | false/f>: Set this parameter flag to false to prevent the wsreversemapping tool from using built-in application identity classes when possible.

  This forces the tool to create custom application identity classes even when there is only one primary key column.
- **-innerIdentityClasses/-inn** <true/t | false/f>: Set this parameter flag to true to have any generated application identity classed be created as static inner classes within the persistent classes.

  The default setting is false.
- **-identityClassSuffix/-is** *<suffix>*: Suffix to append to the class names to form application identity class names, or for inner identity classes, the inner class name.

  The default suffix is Id.
- **-typeMap/-typ** <type mapping>: A string that specifies the default Java classes to generate for each SQL type that is seen in the schema.

  The format is SQLTYPE1=JavaClass1, SQLTYPE2=JavaClass2. The SQL type name first looks for a customization that is based on SQLTYPE(SIZE,PRECISION), then SQLTYPE(SIZE), and then SQLTYPE. If a column with type CHAR is found, it first looks for the CHAR(50,0) type name specification, then it looks for the CHAR(50), and finally for the CHAR. For example, to generate a char array for every char column whose size is exactly 50 characters, and to generate a short for every type name of INTEGER, you might specify, CHAR(50)=char[],INTEGER=short.

Attention: Various databases report different type names differently, one database type might not work for another database. Enable TRACE level logging on the metadata channel to track which type names JPA for WebSphere Application Server is examining.

- **-customizerClass/-cc** *<class name>*: The full class name of an org.apache.openjpa.jdbc.meta.ReverseCustomizer customization plug-in.

  If you do not specify a reverse customizer of your own, the system defaults to a PropertiesReverseCustomizer. This customizer supports specifying simple customization options in the properties file given with the -customizerProperties flag.
  - **-customizerProperties/-cp***<properties file or resource>*: The path or resource name of a properties file to pass to the reverse customizer on initialization.
  - **-customizer/-c** *<property name> <property value>*: The given property name is matched with the corresponding Java bean property in the specified reverse customizer, and set to the given value.

Usage

The wsreversemapping tool is used to perform reverse (bottom-up) mappings of database tables to entity source files. This is useful if developers want to generate Java files from a database for use in other JPA applications. To run this tool:

- You must have database tables and your database connection configured.
- Run the wsreversemapping tool from the command line in the $ {profile_root}/bin directory.
- The tool generates `.java` files for every class, along with an XML descriptor file, `orm.xml`.

The generated Java files from the wsreversemapping tool might require some editing before they can be used in an application. Also, generated files do not contain annotations. Annotations can be added manually. Messages and errors are logged to the administrative console as specified by the configuration.

Examples

Generate entities based on the information saved in the schema.xml file. Schema.xml was created by running the schema tool. The Java files are created in the **src** directory and use the package com.xyz:

```
${profile_root}/bin/wsreversemapping.sh -pkg com.xyz -d ./src schema.xml
```

Generate entities based on information in a DB2 database. Entities are created in the **src** directory, and use the package com.reversemapped:

```
C:\> %profile_root%/bin/wsreversemapping.bat -sa dropDB Magazine.javapkg com.reversemapped -d src
-connectionDriverName=com.ibm.db2.jcc.DB2Driver -connectionURL=jdbc:db2:localhost:50000/TEST
-connectionUser=db2User -connectionPassword=db2Password
```

Additional information

For more information, read the mapping section in the Apache OpenJPA User Guide.

wsschema command

The schema tool can be used to view the database schema in XML form or match an XML schema to an existing database.

The wsschema tool can reflect on the current database schema, optionally translating it into an XML representation for further manipulation. The schema tool can take an XML schema definition, calculate the differences between the XML and the existing database schema, and apply the necessary changes to make the databases correspond to the XML schema. The XML format used by the schema tool is abstract from the differences in SQL dialects used by different vendors. The tool also automatically adapts its SQL to meet foreign dependencies, thus the schema tool is useful as a general way to manipulate the schemas.

Syntax

The command syntax is as follows:

```
wsschema.sh [parameters][arguments]
```

Issue the command from the `bin` subdirectory of the *profile_root* directory.

Parameters

The wsschema tool accepts the standard set of command-line arguments defined by the configuration framework along with the following:

- **-ignoreErrors/-i** <true/t | false/f>: If set to `false`, an exception occurs if the tool encounters any database errors.

  The default is set to `false`.
- **-file/-f** *<stdout / output file>*: Use this option to write a SQL script for the planned schema modifications, rather than committing them to the database.

  When used with the export or reflect actions, the named file is used to write the exported schema XML. If the file names a resource in the class path, data is written to that resource. Use stdout to write to standard output. The default setting is `stdout`.
- **-openjpatables/-ot** <true/t | false/f>: When reflecting the schema, this parameter determines whether to reflect on tables and sequences whose names start with OPENJPA_.

  Certain OpenJPA components can use such tables and sequences, like the table schema factory. When using other actions, openjpaTables controls if these tables can be dropped. The default setting is `false`.
- **-dropTables/-dt** <true/t | false/f>: When this option is set to `true`, schema drops tables that are unused during retain and refresh actions.

  The default is `true`.
- **-dropSequences/-dsq** <true/t | false/f>: If this option is set to `true`, schema drops sequences that are unused during retain and refresh actions.

  The default is `true`.
- **-sequences/-sq** <true/t | false/f>: This flag determines if sequences can be manipulated.

  The default is `true`.
- **-indexes/-ix** <true/t | false/f>: This flag determines if indexes can be manipulated on existing tables.

  The default is `true`.
- **-primaryKeys/-pk** <true/t | false/f>: This flag determines if primary keys can be manipulated on existing tables.

  The default is `true`.
- **-foreignKeys/fk** <true/t | false/f>: This flag determines if foreign keys can be manipulated on existing tables.

  The default is `true`.
- **-record/-r** <true/t | false/f>: This flag permits or prevents writing schema changes made by the schema tool to the current schema factory.

  Select `true` to permit writing schema changes or false to prevent writing schema changes. The default is set to `true`.
- **-schemas/-s** *<schema list>*: Denotes a list of schema and table names the OpenJPA should access when running the schema tool.

  This is the equivalent to setting the openjpa.jdbc.Schemas property to run one time.

  Important: The schema tool accepts the **-action/-a** flag. Multiple actions can be composed in a list, separated by commas. The available actions are:
  - **add**: This is the default action if no other actions are specified. It updated the schema with the given XML documents by adding tables, columns, indexes, or other components. This action never drops any schema components.

- **retain**: This action keeps all schema components in the given XML definition but drops the rest from the database. This action never adds any schema components.
- **drop**: Drops all schema components in the schema XML. This action drops tables only if they would have 0 columns after dropping all columns listed in the XML.
- **refresh**: This action is the equivalent of the **retain** and the **add** functions.
- **build**: Generates SQL to build a schema matching the one in the supplied XML file. Unlike the**add** action, this option does not take into account the fact that part of the schema defined in the XML file may already exist in the database. This action is typically used with the **-file/-f** parameter flag to write a SQL script. This script can be used later to recreate the schema in the XML.
- **reflect**: Generates an XML representation of the current database schema.
- **createDB**: This action generates SQL to recreate the current database. This action is typically used with the **-file/-f** parameter flag to write a SQL script that can be used to recreate the current schema on a new database.
- **dropDB**: Generates SQL to drop the current database. Like the **createDB** action, this can be used with the **-file/-f** parameter flag to script a database drop rather than manually perform it.
- **import**: Imports the given XML schema definition into the current schema factory.

  Note: This action does nothing if the schema factory does not store a record of the schema.
- **export**: Exports the current schema factory stored schema definition to an XML file.

  Note: This can produce an empty file if the schema factory does not store a record of the schema.
- **deleteTableContents**: This action implements SQL to delete all rows from all tables that OpenJPA finds.

Usage

The wsschema tool is used to obtain an XML file that describes the schema of your database. To generate an XML schema file:

- You must have database tables and your database connection configured.
- Run the wsschema tool from the command line in the $ {profile_root}/bin directory.
- The tool generates an XML file that describes the database schema.

Messages and errors are logged to the administrative console as specified by the configuration.

Examples

Add the necessary schema components to the database to match the given XML document without dropping any data:

```
$ wsschema.sh targetSchema.xml
```

Repeat the same action as the previous example, this time not changing the database but instead writing any planned changes to a SQL script:

```
 wsschema.sh -f script.sql targetSchema.xml
```

Write an SQL script that recreates the current database:

```
$ wsschema.sh -a createDB -f script.sql
```

Refresh the schema and delete all the contents of all the tables that OpenJPA knows about:

```
$ wsschema.bat -a refresh,deleteTableContents
```

Drop the current database:

```
$ wsschema.sh -a dropDB
```

Write an XML representation of the current schema to the file *schema.xml*:

```
$ wsschema.sh -a reflect -f schema.xml
```

Additional information

For more information read the JDBC information in the Apache OpenJPA documentation.

wsdbgen command

The command supports utilization of the pureQuery feature in Java Persistence API (JPA) applications.

This command has been renamed wsdbgen for Feature Pack for OSGi Applications and JPA 2.0 and later releases. The command is used in the same way as the wsdb2gen command. The command, wsdb2gen, which implies for DB2 only, works only for DB2 database in WebSphere Application Server V7.0. In this release, the command can be used for DB2, Informix and Oracle databases. Because of that, a synonym, wsdbgen command, is introduced.

The JPA commands (.bat on Windows or .sh on UNIX) are run from the *profile_root*/`bin` directory, rather than from the *app_server_root*/`bin` directory to make sure that you have the latest version of the commands for your release.

Syntax

The command syntax is as follows:

```
wsdbgen.sh [parameters]
```

Before running the command, your `persistence.xml` file must be in the META-INF directory and the META-INF directory must be in the class path.

Parameters

- **`-help`**: This parameter displays the help information.
- **`-pu`**: The name of the persistence unit defined in `persistence.xml` file.
- **`-collection`**: The collection-id which is assigned to package names. The default is `NULLID`.
- **`-url`**: The URL of the target database.

  This is used to validate the generated SQL. A URL must be specified either in the `persistence.xml` file or as a command option. If both are specified, the URL that is specified in the command option is used.
- **`-user`**: The user ID.
- **`-pw`**: The corresponding password to connect to target database.

  If this parameter is not specified, the value found in the `persistence.xml` file is used.
- **`-package`**: If this parameter is specified, the **`-package`** parameter takes the string value package name and a single database package with the specified name is generated. If the **`-package`** parameter is not specified, then one package is generated for each entity class. The entity name is used as package name if the **`-package`** option is not specified. The name length limit is the database limit -1, for example: 128 - 1 = 127.

Usage

The `persistence.xml` file must be included in the application Java archive (JAR) file and is also used as input in the DB2 bind to create the DB2 package. The command requires a connection to a database to validate generated SQL. The database does not have to be the same as the run time database, but it should be at the same version and release level.

Ensure that the following JAR files are on the class path:

- `pdq.jar`
- `pdqmgmt.jar`
- `db2jcc.jar`
- `db2jcc_licence_cu.jar`

If the database URL specifies a DB2 for z/OS database, then the following JAR file must also be on the class path: `db2jcc_licence_cisuz.jar`

Attention: Read more about the DB2 JAR level compliance for IBM Optim PureQuery Runtime at the IBM Support website: System requirements for IBM Optim PureQuery Runtime for Linux, UNIX, and Windows.

Attention: You can review information about pureQuery StaticBinder in the Data Studio Information Center by reading the topic, “The pureQuery StaticBinder utility.”

Examples

DB2

```
wsdbgen.sh  -pu payroll -collection prod1 -url jdbc:db2://myhostname:50000/proddb  -user produser -pw secret
```

Informix

```
wsdbgen.sh  -pu payroll -collection prod1 -url jdbc:ids://myhostname:9089/proddb  -user produser -pw secret
```

ANT task WsJpaDBGenTask

The ANT task **WsJpaDBGenTask** provides an alternative to the **wsdbgen** command.

The **WsJpaDBGenTask** ANT task utility supports utilizing the pureQuery feature in Java Persistence API (JPA) 2.0 and later applications that do not use DB2 databases. Instead of using the **wsdbgen** from the command line, you can use the example code in your ANT build XML file to use the **WsJpaDBGenTask** in your build process.

Both the PDQ runtime Java archive (JAR) files, `pdq.jar` and `pdqmgmt.jar`, must be specified using the ANT -lib option.

Attention: Read more about the DB2 JAR level compliance for pureQuery at the IBM Support website: System requirements for IBM Optim pureQuery Run time for Linux, UNIX, and Windows.

Example

The following example is run in the Windows environment with the ANT command:

```
C:\jpa\ant jar -noclasspath -lib  c:/was8/dev/JavaEE/j2ee.jar
-lib ${app_server_root}/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar
-lib c:/sqllib/java/db2jcc.jar
-lib c:/sqllib/java/db2jcc_license_cu.jar
-lib c:/sqllib/java/pdq.jar
-lib c:/sqllib/java/pdqmgmt.jar
```

When calling the ANT command, the JAR files for pureQuery, JPA, and the JDBC driver must be on the library list.

```
<?xml version="1.0"?>
<project name="sample" default="jar">
<taskdef name="enhancer" classname="org.apache.openjpa.ant.PCEnhancerTask" />
<taskdef name="wsdbgen" classname="com.ibm.websphere.persistence.pdq.ant.WsJpaDB2GenTask" />
<target name="clean" description="remove intermediate files">
<delete dir="classes"/>
<delete dir="enhanced" />
```

```
<delete>
<fileset dir="." includes="META-INF/*.pdqxml" />
<fileset dir="." includes="sample.jar" />
</delete>
</target>

<target name="compile"
description="compile the Java source code to class files">
<mkdir dir="classes"/>
<javac srcdir="." destdir="classes">
<classpath>
<pathelement location="c:/was8/dev/JavaEE/j2ee.jar>"
<pathelement location="c:/was8/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar" />
</classpath>
</javac>
</target>

<target name="enhance" depends="compile" >
<mkdir dir="enhanced" />
<enhancer directory="./enhanced" >
<config propertiesFile="META-INF/persistence.xml" />
<classpath>
<pathelement location="." />
<pathelement location="classes" />
</classpath>
</enhancer>
</target>


<target name="wsdbgen" depends="enhance" >
<wsdb2gen pu="MyAntTest" url="jdbc:db2://localhost:50000/demodb" user="user1" pw="secret" >
<classpath>
<pathelement location="."/>
<pathelement location="enhanced" />
</classpath>
</wsdb2gen>
</target>


<target name="jar" depends="wsdbgen"
description="create a Jar file for the application">
<jar destfile="sample.jar">
<fileset dir="classes" includes="**/*.class"/>
<fileset dir="." includes="META-INF/*.xml" />
</jar>
</target>
</project>
```

SQL statement batching for JPA applications

SQL statement batching can improve the performance of your application server.

About this task

Attention: Support for SQL statement batching is no longer a WebSphere Application Server for JPA extension. You can find information regarding statement batching in the Apache OpenJPA documentation.

By default, statement batching is enabled for DB2 and Oracle databases. To enable SQL statement batching and to set the batch limit for JPA applications, you must configure the `persistence.xml` file. The following steps review how to enable and disable statement batching, and set the batch limit:

Procedure

1. Define the UpdateManager property in the `persistence.xml` file. For example:

```
<property name="openjpa.jdbc.UpdateManager"
value="com.ibm.ws.persistence.jdbc.kernel.OperationOrderUpdateManager(batchLimit=100)"/>
```

Note: The example shows that the SQL statement batch limit is set to `100`.

Remember: If you are using a DB2 or an Oracle database, by default the SQL statement batching is enabled and set to `batchLimit=100`. However, if you are using DB2 or Oracle, you are not required to specify this property in the `persistence.xml` file.

2. If you must disable SQL statement batching, set the batchLimit value to `0` or remove the property. However, if you are using a DB2 or an Oracle database, you must specify the DBDictionary property, database, and set the defaultBatchLimit to `0` . For example:

```
<property name="openjpa.jdbc.DBDictionary" value="db2(defaultBatchLimit=0)"/>
```

Results

You have now updated the `persistence.xml` file to enable or disable statement batching and set the batch limit.

Database generated version ID with JPA

Java Persistence API (JPA) for WebSphere Application Server has extended OpenJPA to work with database generated version IDs. These generated version fields, timestamp, or token, can be used to efficiently detect changes to a given row.

Trigger based version ID generation is supported for all databases that WebSphere Application Server supports. Support is based on two Version Strategies in JPA for WebSphere Application Server.

- @VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy"), if the entity version field type is Timestamp, and
- @VersionStrategy("com.ibm.websphere.persistence.RowChangeVersionStrategy"), if the entity version field type is Long

The Entity class is defined with the new Version Strategy annotation. The Entity has a surrogate version column. For example,

```
@Entity(name="Item")
@VersionColumn(name="versionField")
@VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy")
public class Item implements Serializable
{
 @Id
 private int id2;

 private String name;

 private double price;

   @OneToOne
   private Owner master;
}
```

The create table statement is as follows:

```
CREATE TABLE ITEM
(ID2  INT  NOT NULL,
 NAME  VARCHAR(50) ,
 PRICE   DOUBLE,
 OWNER_ID INT,
             VERSIONFIELD GENERATED ALWAYS FOR EACH ROW ON
              UPDATE AS ROW CHANGE TIMESTAMP
         PRIMARYKEY(ID2));
```

During any updates to Item, insert or update, the VersionColumn value is updated in the database. After the update, the value for VersionColumn is retrieved from the database and updated in the in-memory object. Thereby the objects in the data cache reflect the correct version value. Here the Entity is using the @VersionColumn, which produces a Surrogate Version ID rather than defining an explicit field in the entity.

The Entity could also use @Version annotation to define an explicit version field. The explicit version field could be of type Long or Timestamp corresponding to the @VersionStrategy. During any updates to Item, insert or update, the Version value is updated in the database. After the update, the value for Version is retrieved from the database and updated in the in memory object. Thereby the objects in the data cache reflect the correct version value.

This is an example where the Entity has a version field defined, and the type Timestamp matches the RowChangeTimestampStrategy in the @VersionStrategy. If the version field type is using type long, then the RowChangeVersionStrategy should be annotated to match:

```
@Entity(name="Item")
@VersionStrategy("com.ibm.websphere.persistence.RowChangeTimestampStrategy")
public class Item implements Serializable

{
 @Id
 private int id2;

 private String name;

 private double price;

   @Version
   private Timestamp versionField;

   @OneToOne
   private Owner master;
}
```

Note: Be aware of the following conditions when you use RowChangeVersionStrategy:

- For z/OS DB2 V9 and Linux, UNIX, and Windows DB2 V9.5, the generated database column must be of type timestamp, but both the RowChangeTimestampStrategy and the RowChangeVersionStrategy are supported. Microsoft SQL Server only supports a non-timestamp generated version ID that goes with the RowChangeVersionStrategy. To use the RowChangeTimestampStrategy, you must use a trigger on a timestamp field in the database. For other databases, you can use triggers to simulate database version generation and use either strategy.
- For z/OS DB2 V9, install the PTF for APAR PK67706, and ensure that you have installed the required level of IBM Optim PureQuery Runtime (1.3.100 or later) and JCC drivers (3.52.95 or later).

Mapping persistent properties to XML columns for JPA

If your database supports Extensible Markup Language (XML) column types, you can use mapping tools to manage XML objects. You have the choice of mapping XML columns to a Java string or a Java byte array field.

About this task

Attention: Support for mapping persistent properties to XML columns is no longer a WebSphere Application Server for JPA extension. You can find information regarding XML mapping in the Apache OpenJPA documentation.

JPA for the application server supports the management of XML objects by using a third-party solution for mapping management. These mapping techniques make using the XML objects as strings or byte arrays difficult.

DB2, Oracle, and SQLServer databases support XML column types, XPath queries, and indices over these columns.

Persistent properties to XML mapping

An embedded class with XML column support must use XML marshaling to write the data to the XML column and unmarshalling to retrieve the data from the XML column. The path expressions and predicates over the embedded class are converted to XML predicates, XPATH expressions, or XQuery expressions and are written to the database.

WebSphere Application Server supports JPA applications to use a third-party tool for XML mapping. This is done through the extension points for custom field mappings. The third-party mapping tool uses the extension points by providing a custom value handler for the persistent fields that are mapped to the XML columns. In OpenJPA, this value handler is named org.apache.openjpa.xmlmapping.XmlValueHandler and this handler requires the @Strategy annotation on the Java field that is mapped to the XML column.

Procedure

1. Annotate the entity property using the XML value handler strategy. The mapping of persistent properties to XML columns requires the @Strategy and the @Persistent annotation.

   ```
   @Persistent
   @Strategy("org.apache.openjpa.xmlmapping.XmlValueHandler")
   ```

   The XML value handler for the persistent property is set to org.apache.openjpa.xmlmapping.XmlValueHandler.
2. Change the default for fetch type if it is necessary. For example:

   ```
   @Persistence(fetch=FetchType.LAZY)
   ```

   The fetch type is now LAZY. If a value for the fetch type is not entered, the default is set to EAGER.
3. Annotate your embedded classes with the binding annotations for Java API for XML Binding (JAXB). These bindings can be created from an XML schema by using the Java Architecture for XML Binding Compiler (XJC).
4. Make sure that the class that maps to the root of the XML document is annotated with @XmlRootElement, in addition to the other annotations.
5. Compile your Java sources.
6. Run the enhancer tool on the entities. Refer to the topic on the entity enhancer tool for more information.

Example

For example, shipAddress, a property of Order Entity, is mapped to XML column shipaddr:

```
@Entity
public class Order {
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
  int oid;
 @Persistent
 @Strategy("org.apache.openjpa.xmlmapping.XmlValueHandler")
      @Column(name="shipaddr")
      Address shipAddress;
...
```

The OpenJPA mapping tool generates a SHIPADDR column with XML type in the table definition for ORDER table.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

  The corresponding product variable is *smpe.install.root*.

  The default is /usr/lpp/zWebSphere/V8R5.

Assembling applications that use the Java Persistence API

Assembling a JPA application in a Java EE environment

You have developed and configured your applications to work with the Java Persistence API (JPA). Now you need to package the JPA applications for your environment.

About this task

Procedure

Package the application. There are several packaging options for an application that uses JPA in a Java EE environment. Choose the packaging option that best suits the JPA usage and configuration within the

modules of your application. These are some of the most common packaging options. For a definitive list of packaging options, see the Java Persistence API specification.

Note: IBM Optim PureQuery Runtime, add the *persistence_unit_name*.pdqxml file to the JPA application JAR file. The files are located in same META-INF directory where the persistence.xml file is located. These persistence files were created during the development task. See the topic, Developing JPA applications for a Java EE environment for more information.

- For a standalone Enterprise JavaBeans (EJB) module or a stand-alone application client module, package the EJB and application client modules in a standard Java archive (JAR) file. Ensure that you package the application with these conditions:
  - The JAR file must contain your EJB class files or the Java class files for the application client.
  - The META-INF directory of the archive must include your persistence.xml file.
  - If your application uses mapping files, orm.xml, or a custom mapping file, the JAR file must also contain those files. If the location of the orm.xml file is not specified in the persistence unit, the default location is the META-INF directory of the JAR file.
  - The Criteria Metamodel class files that are generated by the Annotation Processor when you developed your application needs to be included in the JAR file in the same location as the entity class files.
- For a stand-alone web module, package the application in a standard web application archive (WAR) file. Ensure that you package the application with these conditions:
  - The Criteria Metamodel class files that are generated by the Annotation Processor while developing your application needs to be included in the WAR file in the same location as the entity class files.
  - The WAR file must contain your web application class files. The web application class files must be included in the WEB-INF/classes directory or in a JAR file that is located in the WEB-INF/lib directory of the WAR file.
  - The persistence.xml file must be included in the WEB-INF/classes/META-INF directory or in the META-INF directory of a JAR file that is included in your WEB-INF/lib directory of your WAR file.
  - If your application uses mapping files, orm.xml, or a custom mapping file, the WAR file must also contain those files. Mapping files can reside in the WEB-INF/classes directory or in a JAR file that is contained within the WEB-INF/lib directory of the WAR file. Use the <mapping-file> element of the persistence.xml file to specify the location of mapping files. For example:

    ```
    <mapping-file>META-INF/JPAorm.xml</mapping-file>
    ```
- For enterprise application that contains one or more modules, package the application in a standard Enterprise application archive (EAR) file. An enterprise application can contain one ore more EJB module, web module, or application client module. Ensure that you package the application with these conditions:
  - If multiple modules use the same persistence unit, you can create a persistence archive and package the persistence archive within the EAR file.
  - Include your entity classes, associated Criteria Metamodel classes, any necessary supporting classes, the persistence.xml file, and additional mapping files in the persistence archive file. Follow the packaging rules for EJB and application client modules for the location of your persistence.xml file and mapping files.
  - Each module that uses the persistence archive must have a class path entry in its META-INF/MANIFEST.MF file. Here is an example manifest file:

    ```
    Manifest-Version: 1.0
    Class-Path: MyJPAEntities.jar
    ```
  - If your modules use separate persistence units and share entity classes, you can package the entity classes in a persistence archive and specify different persistence.xml file and mapping files for each module. If the modules do not share entity classes or a persistence configuration, package each module as a standalone EJB module, a stand-alone application client module, or a standalone web application archive and then package them in the EAR file.

What to do next

For more information about the commands, classes or other OpenJPA information, refer to the Apache OpenJPA User Guide.

Assembling JPA applications for a Java SE environment

You have developed and configured your applications to work with the Java Persistence API (JPA). Now you must package the JPA applications for your environment.

About this task

For this task, you must specify the `com.ibm.ws.jpa.thinclient_8.0.0.jar` stand-alone Java archive (JAR) file in your class path. This stand-alone JAR file is available from the client and server installation images. The location of this file on the client installation image is ${*app_client_root*}/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar. The location of this file on the server installation image is ${*app_server_root*}/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jar.

Procedure

1. Package the application.

   Note: Package the persistence units in separate JAR files to make them more accessible and reusable. If you package the persistence units this way, they can be tested outside the container both with and without the occurrence of database persistence. The persistence units can be included in stand-alone applications or they can be packaged into EAR files as persistence archive files. If you package the persistence unit into a persistence archive file, all of the application components must be able to access the persistence archive. The application that uses the persistence units must declare a dependency on the persistence archive using the `MANIFEST.MF Class-Path:` declaration.

   The Criteria Metamodel class files that are generated by the Annotation Processor when you developed your JPA application, must be included in the JAR file in the same location as the entity class files.

   Note: If you are using IBM Optim PureQuery Run time, add the *persistence_unit_name*`.pdqxml` files to the JPA application JAR file. The files are located in same `META-INF` directory where your `persistence.xml` file is located. These persistence files were created during the development task, Developing JPA applications for a Java SE environment.

   To package the application use the following command:

   ```
   jar -cvf ${jar_Name} ${entity_Path}
   ```

   where ${*jar_Name*} represents the name of the JAR file to create, and ${*entityPath*} represents the root location where the entities reside, which is where you compiled them. Make sure that your ${`entity_Path`} also contains the `META-INF/persistence.xml` file.

2. When you run your stand-alone application, specify the JAR files in your class path when starting your application. The JPA runtime starts the stand-alone JAR file, `com.ibm.ws.jpa.thinclient_8.0.0.jar`. For example, use the following Java call to run the com.xyz.Main stand-alone application:

   ```
   java -cp /your/directory/${jar_Name}
   -javaagent:${app_client_root}/runtimes/com.ibm.ws.jpa.thinclient_8.0.0.jarcom.xyz.Main
   ```

What to do next

For more information about any of the commands, classes or other OpenJPA information, refer to the Apache OpenJPA User Guide.

Using JPA access intent

Java Persistence API (JPA) access intent specifies the isolation level and lock level used when reading data from a data source. Access intent controls the Java Database Connectivity (JDBC) isolation level and whether read, update, or exclusive locks are acquired when retrieving data.

About this task

For a JPA persistence provider on the application server, the application can specify isolation and ReadLockMode based on a TaskName. The TaskName provides a better control over applying these characteristics. The application defines a set of entity types and corresponding access intent for each TaskName defined in a persistence unit.

Restriction:

- Access intent is available for application in the Java EE server environment
- Access intent is applicable to non-query entity manager interface methods. Query uses query hint interface to set its isolation and read lock values.
- Access intent is only available for DB2 databases.
- Access intent is in effect only when pessimistic lock manager is used. Add the following to the persistence unit property list. <property name="openjpa.LockManager" value="pessimistic"/>

Table 73. Access intent Properties and Descriptions. The following table compares the Enterprise JavaBeans (EJB) 2.x entity bean access intent with the JPA access intent properties:

WebSphere EJB 2.x entity bean access intent	JPA access intent	Description
optimistic	isolation: Read Committed	Data is read but no lock is held. Version ID is used on update to ensure data integrity. Other transactions can read and update data.
	lockManager: Optimistic	
	query Hint: ReadLockMode: READ	
pessimistic read	isolation: Repeatable Read	Data is read with shared locks. Other transactions attempting to update data are blocked.
	lockManager: Optimistic	
	query Hint: ReadLockMode: READ	
pessimistic update	isolation: Repeatable Read	Data is retrieved with update or exclusive lock. Other writes are blocked until commit. This access intent can be used to serialize update access to data when there are multiple writers.
	lockManager: Pessimistic	
	query Hint: ReadLockMode: WRITE	
pessimistic exclusive	isolation: Serializable	Data is retrieved with update or exclusive lock. Other writes are blocked until commit. This access intent can be used to serialize update access to data when there are multiple writers.
	lockManager: Pessimistic	
	query Hint: ReadLockMode:WRITE	

A TaskName is set on a transaction context by one of the following:

- TaskName is automatically set in the EJB container when a transaction begins using WebSphere local transaction (EJB unspecified transaction), JTA global transaction in a Container-Managed Transaction (CMT) or user-initiated global transaction in a Bean-Managed Transaction (BMT).
- TaskName is manually set in an application using the TaskNameAccessor API provided for JPA.

Using task names supports the specification of access intent on a request scope rather than specifying it in the `persistence.xml` file, which has an application scope across all entities. Often a query is contained

in a method or component which is used in many different transaction contexts. Some of these contexts might require repeatable-read and update lock intent but other contexts do not.

Isolation level and read locks can be specified on:

- An application scope in the `persistence.xml` file. These isolation levels and read lock types are properties specified in the `persistence.xml` file. They apply to all entities that are defined in the persistence unit.
- A transaction scope in the task name. Transaction scoped hints override application scope values.
- Query instance with a query hint. Query hint can be used to override isolation and ReadLockMode for a particular query instance. A query hint overrides isolation level and read locks specified at the application or transaction scope.

Procedure

1. "Setting a TaskName using TaskNameAccessor API" This task explains how to use the TaskNameAccessor API to set JPA TaskName at run time.
2. "Specifying TaskName in a JPA persistence unit" on page 521 This task explains how to specify a TaskName in JPA persistence unit .

What to do next

For more information about Access intent, see the topic, Access intent service.

Setting a TaskName using TaskNameAccessor API

Using the TaskNameAccessor API to set Java Persistence API (JPA) TaskName at runtime.

About this task

In the Enterprise JavaBeans (EJB) container, a task name is automatically set by default upon a transaction begins. This action is performed when a component or business method is invoked in a CMT session bean or when an application invoke the `sessionContext.getTransaction().begin()` in a BMT session bean. This TaskName consists of a concatenation of the fully package qualified session bean type, a dot character and the method name. For example: `com.acme.MyCmtSessionBean.methodABC`.

If using JPA in the context of the web container, an application must use the TaskNameAccessor API to set the TaskName in the current thread of execution.

Note: Once a TaskName is set on a transaction context, application must not set the TaskName again in the same transaction. This will avoid problems with on the JDBC connection for different database access.

This example contains the TaskNameAccessor API definition

```
package com.ibm.websphere.persistence;

public abstract class TaskNameAccessor {

 /**
  * Returns the current task name attached in the current thread context.
  * @return current task name or null if none is found.
  */
 public static String getTaskName ();

 /**
  * Add a user-defined JPA access intent task name to the current thread
  * context.
  *
  * @param taskName
  * @return false if an existing task has already attached in the current
```

```
  *         thread or Transaction Synchronization Registry (TSR) is not
  *         available (i.e. in JSE environment).
  */
 public static boolean setTaskName(String taskName);

}
```

This code example shows how to set a TaskName using TaskNameAccessor.

```
package my.company;

@Remote
class Ejb1 {
    // assumer no tx from the caller
    @TransactionAttribute(Requires)
    public void caller_Method1() {

        // an implicit new transaction begins
        // TaskName "my.company.Ejb1.caller_Method1" set on TSR

        ejb1.callee_Method?();
    }

    @TransactionAttribute(RequiredNew)
    public void callee_Method2() {

        // an implicit new transaction begins i.e. TxRequiredNew.
        // TaskName "my.company.Ejb1.callee_Method2" set on TSR
    }

    @TransactionAttribute(Requires)
    public void callee_Method3() {

        // In caller's transaction, hence TaskName remains
        //     "my.company.Ejb1.caller_Method1"
    }

    @TransactionAttribute(NotSupported)
    public void callee_LocalTx () {

        // Unspecified transaction, a new local transaction implicitly started.
        // TaskName "my.company.Ejb1.callee_LocalTx" set on TSR
    }

}
```

Attention: In the above example, an application must be aware of transaction boundary will be subtly changed if Ejb1 uses local interface (@Local). For example, when `caller_Method1()` calls `callee_Method3` or `callee_LocalTx`, this will be treated as a Java method call. No EJB transaction semantics are honored.

What to do next

Once you have completed this step, continue on with the topic Specify TaskName in a JPA persistence unit.

Specifying TaskName in a JPA persistence unit

Specifying a TaskName in Java Persistence API (JPA) persistence unit

About this task

A TaskName is defined in the `persistence.xml` file using the `wsjpa.AccessIntent` property name in a persistence unit. The property value is a list of TaskNames, entity types and access intent definitions. The following example shows the contents of the `wsjpa.AccessIntent` property name in a persistence unit.

```
<property name  = "wsjpa.AccessIntent"
         value = "Tasks=' <taskName> { <entityName> ( <isolationLockValue> ) } ' "/>
                        A            A              A                    | | |
                        |            |              +--------- , --------+ | |
                        |            +------------------ , -----------------+ |
                        +----------------------- , --------------------------+
  Tasks                       ::= <task> [ ',' <task> ]*

  <task>                      ::= <taskName> '{' <entity> [ ',' <entity> ]* '}'

  <entity>                    ::= <entityName> '(' <isolationLockValues> ')'

  <taskName>                  ::= <fully_qualified_identifier>

  <entityName>                ::= <fully_qualified_identifier>

  <fully_qualified_identifier> ::= <identifier> [ '.' <identifier> ]*

  <identifier>                ::= <idStartCharacter> [ <idCharacter> ]*

  <idStartCharacter>          ::= Character.isJavaIdentifierStart | '?' | '*'

  <idStartCharacter>          ::= Character.isJavaIdentifierPart  | '?' | '*'

  <isolationLockValues>       ::= <isolationLockValue> [ ',' <isolationLockValue> ]

  <isolationLockValue>        ::= <isolation> | <readLock>

  <isolation>                 ::= "isolation" '=' <isolationValue>

  <readLock>                  ::= "readlock" '=' <readlockValue>

  <isolationValue>            ::= "read-uncommitted"|"read-committed"|"repeatable-read"|"serializable"

  <readlockValue>             ::= "read" | "write"
```

Before setting the TaskName in a persistence unit, keep the following in mind:

- White spaces are ignored between tokens.
- Only <isolation> and <readLock> contents are not case sensitive.
- <TaskName> is in the form of a fully package qualified method name, such as com.acme.bean.MyBean.increment, or an arbitrary user-defined task name, such as MyProfile.
- <entityName> is in the form of a fully package qualified class name such as com.acme.bean.Entity1.
- The wild card characters '?' or '*' can be used in <TaskName> and <entityName>. "?" matches any single character and "*" matches zero or more sequence characters.
- Only hintNames isolation and readLock are allowed on a task definition and the order is not significant
- If readLock has the value `write`, then isolation must be `repeatable-read` or `serializable`
- If readLock has the value `read`, it has no effect if the isolation is `read-uncommitted`.

The following code example shows how to specify a TaskName in JPA persistence unit.

```
package my.company;

@Remote
class Ejb1 {
    // assumer no tx from the caller
    @TransactionAttribute(Requires)
    public void caller_Method1() {

        // an implicit new transaction begins
        // TaskName "my.company.Ejb1.caller_Method1" set on TSR

        ejb1.callee_Method?();
    }
```

```
    @TransactionAttribute(RequiredNew)
    public void callee_Method2() {

        // an implicit new transaction begins i.e. TxRequiredNew.
        // TaskName "my.company.Ejb1.callee_Method2" set on TSR
    }

    @TransactionAttribute(Requires)
    public void callee_Method3() {

        // In caller's transaction, hence TaskName remains
        //     "my.company.Ejb1.caller_Method1"
    }

    @TransactionAttribute(NotSupported)
    public void callee_LocalTx () {

        // Unspecified transaction, a new local transaction implicitly started.
        // TaskName "my.company.Ejb1.callee_LocalTx" set on TSR
    }

}
```

Since a wild card can be used to specify TaskName and entity type, multiple specification matches may occur at runtime. The order defined in the `wsjpa.AccessIntent` property will be used to search for task names and entity types.

```
<properties>
  <property name="wsjpa.AccessIntent" value="Tasks="
        *.Task1 { *.Employee1 ( isolation=read-uncommitted                ),
                  *.Employee? ( isolation=repeatable-read,  readlock=write ),
                },
        *       { *.Employee3 ( isolation=serializable,     readlock=write ) },
      '" />
</properties>
```

Associating persistence providers and data sources

Java Persistence API (JPA) applications specify the underlying data source that is used by the persistence provider to access the database.

About this task

The application server provides three methods for defining the data sources in the `persistence.xml` file.

Procedure

- Explicitly specify the Java Naming and Directory Interface (JNDI) name in the `persistence.xml` file, and the application directly references the data source. Switching to another data source requires an update to the `persistence.xml` file.

  JPA has two transactional patterns for accessing a data source:

  – The Java Transaction API (JTA) resource pattern depends on global transactions. The JTA resource pattern is typically used within the scope of an Enterprise JavaBeans (EJB) session facade. This supports the session bean to control transaction and security contexts while JPA handles the persistence mappings. In this case, the application does not use the EntityTransaction interface but relies on the EntityManager enlisted with the global transaction when it is accessed.

  – The non-JTA resource pattern is used when dealing with a single resource in the absence of global transactions. The non-JTA resource pattern is typically used within the scope of a web application or an application client. The application controls the transaction with the data source with the EntityTransaction interface.

Within the application server, the use of the <non-jta-data-source> element requires a special configuration for a non-transactional data source. Data sources that are configured for the application server do not function as a <non-jta-data-source>, because all data sources that are configured by the application server are automatically enlisted with the current transactional context. To prevent this automatic enlistment, add an additional data source custom property `nonTransactionalDataSource=true`:

1. Select **Resources** > **JDBC** > **Data sources**
2. Select the name of the data source that you want to configure.
3. Select **WebSphere Application Server data source properties** from the **Additional Properties** heading.
4. Select **Non-transactional data source**.
5. Click **OK**.

Note: The JPA specification assumes that connections are obtained with an isolation level that does not hold long-term locks in the database, such as `READ_COMMITTED`. This might not match the WebSphere Application Server default isolation level, which is `REPEATABLE_READ` for most databases. You can find the level that is used for your database by reading the topic, Requirements for setting isolation level.

If the default for your database is not `READ_COMMITTED`, you can change the default by adding an additional data source custom property webSphereDefaultIsolationLevel.

Table 74. Isolation level values. This table shows valid isolation level values.

Value	Isolation Level
1	READ_UNCOMMITTED
2	READ_COMMITTED (JPA default)
4	REPEATABLE_READ (WebSphere Application Server default)
8	SERIALIZABLE

If the isolation level is set to a value that holds long-term read locks, configure the JPA provider to use Pessimistic Locking instead of the default Optimistic Locking. For the JPA provider included with WebSphere Application Server, you can do this by adding the following properties to `persistence.xml` file:

```
<property name="openjpa.Optimistic" value="false"/>
<property name="openjpa.LockManager" value=pessimistic"/>
```

The JPA specification mandates that the data sources that are defined in `<jta-data-source>` and `<non-jta-data-source>` elements of a persistence unit register in the JNDI name space. For example, the `persistence.xml` file should contain an entry like the following:

```
<jta-data-source>jdbc/DataSourceJNDI</jta-data-source>
```

- The JPA for WebSphere Application Server solution extends the JNDI data-source implementation to allow you to reference data sources in the component name space. In the EJB or web module deployment descriptor file, this is the <resource-ref> element. You can prefix the data source with java:`comp/env/` so the application indirectly references the data source by using the local JNDI name. In this association, the application does not require updates, you change <resource-ref> to use another data source. See the following example:

  ```
  <jta-data-source>java:comp/env/jdbc/DataSourceJNDI</jta-data-source>
  ```

- You can declare openjpa.Connection* properties in the persistence unit as follows:

  ```
  <property name="openjpa.ConnectionDriverName" value="org.apache.derby.jdbc.EmbeddedDriver" />
  <property name="openjpa.ConnectionURL" value="jdbc:derby:target/database/jpa-test-database;create=true"/>
  ```

  OR

  You can use alternative standard JPA properties that are equivalent to the OpenJPA properties, such as:

Table 75. Standard JPA 2.0 property equivalents. Standard JPA 2.0 properties and the OpenJPA equivalents.

Standard JPA 2.0	OpenJPA Equivalent
javax.persistence.jdbc.driver	openjpa.ConnectionDriverName

Table 75. Standard JPA 2.0 property equivalents (continued). Standard JPA 2.0 properties and the OpenJPA equivalents.

Standard JPA 2.0	OpenJPA Equivalent
javax.persistence.jdbc.url	openjpa.ConnectionURL

What to do next

For information about configuring data sources, see the topic on creating and configuring a data source.

For information about data sources and JPA, see the section on persistence in the Apache OpenJPA User Guide.

Chapter 11. Developing Internationalization service

This page provides a starting point for finding information about globalization and the internationalization service, a WebSphere extension for improving developer productivity.

With the internationalization service, you can automatically recognize the time zone and location information of the calling client so that your application can act appropriately. The technology enables you to deliver each user, around the world, the right date and time information, the appropriate currencies and languages, and the correct date and decimal formats.

This documentation also includes information about internationalizing interface strings using the localizable-text application programming interface.

Task overview: Globalizing applications

An application that can present information to users according to regional cultural conventions is said to be *globalized*: The application can be configured to interact with users from different localities in culturally appropriate ways. In a globalized application, a user in one region sees error messages, output, and interface elements in the requested language. Date and time formats, as well as currencies, are presented appropriately for users in the specified region. A user in another region sees output in the conventional language or format for that region. Globalization consists of two phases: *internationalization* (enabling an application component for multicultural support) and *localization* (translating and implementing a specific regional convention). This product supports globalization through the use of its localizable-text API and internationalization service.

Procedure

- Make sure the server runtime environment is properly configured.

  For more information about supported locales and character encodings, see “Working with locales and character encodings” on page 529.
- Implement message catalogs in your application by using the localizable-text API.

  This product supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

  For more information about the localizable-text API, see “Task overview: Internationalizing interface strings (localizable-text API)” on page 531.
- Implement more extensive locale support by using the internationalization service.

  With the internationalization service, you can manage the distribution of the internationalization information, or *internationalization context*, that is necessary to support globalized Java Platform, Enterprise Edition (Java EE) application components. Supported application components also include web service client environments and web service-enabled enterprise beans.

  For more information about the internationalization service, see “Task overview: Internationalizing application components (internationalization service)” on page 542.

Globalization

An application that can present information to users according to regional cultural conventions is said to be *globalized*: The application can be configured to interact with users from different localities in culturally appropriate ways. In a globalized application, a user in one region sees error messages, output, and interface elements in the requested language. Date and time formats, as well as currencies, are presented appropriately for users in the specified region. A user in another region sees output in the conventional language or format for that region. Globalization consists of two phases: *internationalization* (enabling an application component for multicultural support) and *localization* (translating and implementing a specific regional convention).

Historically, the creation of globalized applications has been restricted to large corporations writing complex systems. However, given the rise in distributed computing and in the use of the World Wide Web, application developers are pressured to globalize a much wider variety of applications. This trend requires making globalization techniques much more accessible to application developers.

Internationalization of an application is driven by two variables, the time zone and the locale. The *time zone* indicates how to compute the local time as an offset from a standard time like Greenwich Mean Time. The *locale* is a collection of information about language, currency, and the conventions for presenting information like dates. A time zone can cover many locales, and a single locale can span time zones. With both time zone and locale, the date, time, currency, and language for users in a specific region can be determined.

By convention, a given locale is specified with a pair of codes (for language and region) that are governed by different standards. The ISO-639 standard governs the language code; the ISO-3166 standard governs the regional code. In notation, the two codes are typically joined by an underscore (_) character, for example, en_US for English in the United States. In Java code, locales are set and retrieved by means of the java.util.Locale class.

A first step: Localization of interface strings

In an application that is not globalized, the user interface is unalterably written into the application code. Internationalizing a user interface adds a layer of abstraction into the design of an application. The additional layer of abstraction enables you to localize the application for each locale that must be supported by the application.

In a localized application, the locale determines the message catalog from which the application retrieves message strings. Instead of printing an error message, the application represents the error message with some language-neutral information; in the simplest case, each error condition corresponds to a key. To print a usable error message, the application looks up the key in a *message catalog*. Each message catalog is a list of keys with associated strings. Different message catalogs provide strings for the different languages that are supported. The application looks up the key in the appropriate catalog, retrieves the corresponding error message in the requested language, and prints the string for the user.

Localization of text can be used for far more than translating error messages. For example, by using keys to represent each element in a graphical user interface (GUI) and by providing the appropriate message catalogs, the GUI (buttons, menus, and so on) can support multiple languages. Extending support to additional languages requires that you provide message catalogs for those languages; in many cases, the application needs no further modification.

The localizable-text package is a set of Java classes and interfaces that can be used to localize the strings in distributed applications easily. Language-specific string catalogs can be stored centrally so that they can be maintained efficiently.

Globalization challenges in distributed applications

With the advent of Internet-based business computational models, applications increasingly consist of clients and servers that operate in different geographical regions. These differences introduce the following challenges to the task of designing a solid client-server infrastructure:

Clients and servers can run on computers that have different endian architectures or code sets

Clients and servers can reside in computers that have different endian architectures: A client can reside in a little-endian CPU, while the server code runs in a big-endian one. A client might want to call a business method on a server running in a code set different from that of the client.

A client-server infrastructure must define precise endian and code-set tracking and conversion rules. The Java platform has nearly eliminated these problems in a unique way by relying on its

Java virtual machine (JVM), which encodes all of the string data in UCS-2 format and externalizes everything in big-endian format. The JVM uses a set of platform-specific programs for interfacing with the native platform. These programs perform any necessary code set conversions between UCS-2 and the native code set of a platform.

Clients and servers can run on computers with different locale settings

Client and server processes can use different locale settings. For example, a Spanish client might call a business method upon an object that resides on an American English server. Some business methods are locale-sensitive in nature; for example, given a business method that returns a sorted list of strings, the Spanish client expects that list to be sorted according to the Spanish collating sequence, not in the English collating sequence of the server. Because data retrieval and sorting procedures run on the server, the locale of the client must be available to perform a legitimate sort.

A similar consideration applies in instances where the server has to return strings containing date, time, currency, exception messages, and so on, that are formatted according to the cultural expectations of the client.

Clients and servers can reside in different time zones

Client and server processes can run in different time zones. To date, all internationalization literature and resources concentrate mainly on code set and locale-related issues. They have generally ignored the time zone issue, even though business methods can be sensitive to time zone as well as to locale.

For example, suppose that a vendor makes the claim that orders received before 2:00 PM are processed by 5:00 PM the same day. The times given, of course, are in the time zone of the server that is processing the order. It is important to know the time zone of the client to give customers in other time zones the correct times for same-day processing.

Other time zone-sensitive operations include time stamping messages logged to a server, and accessing file or database resources. The concept of Daylight Savings Time further complicates the time zone issue.

Java Platform, Enterprise Edition (Java EE) provides support for application components that run on computers with differing endian architecture and code sets. It does not provide dedicated support for application components that run on computers with different locales or time zones.

The conventional method for solving locale and time zone mismatches across remote application components is to pass one or more extra parameters on all business methods needed to convey the client-side locale or time zone to the server. Although simple, this technique has the following limitations when used in Enterprise JavaBeans (EJB) applications:

- It is intrusive because it requires that one or more parameters be added to all bean methods in the call chain to locale-sensitive or time zone-sensitive methods.
- It is inherently error-prone.
- It is impracticable within applications that do not support modification, such as legacy applications.

The internationalization service addresses the challenges posed by locale and time zone mismatch without incurring the limitations of conventional techniques. The service systematically manages the distribution of internationalization contexts across the various components of EJB applications, including client applications, enterprise beans, and servlets. For more information, see “Task overview: Internationalizing application components (internationalization service)” on page 542.

Working with locales and character encodings

Internationalization support for this product relies on that provided by the Java Platform, Standard Edition (JSE). Support varies by platform.

Procedure

- Verify that the operating system on which the application server is installed supports the locales and encodings that you plan to use.

  Java internationalization support might use underlying services of the operating system. For example, if user IDs for your server are expected to contain non-English characters, make sure that the operating system is configured to process those characters.
- Plan for encoding changes as necessary.

  Consider differences in encoding support among operating system subcomponents. Although this product and the Java platform are based on Unicode encoding, it is not always possible to run applications in a purely Unicode environment.
- Set the console.encoding property as necessary.

Results

If your application produces an UnsupportedEncodingException exception, check your operating system documentation to determine if the target operating system supports the required encoding and adjust the runtime environment as needed. Use the converter.properties file as appropriate to map an unsupported character set to a supported character set. A typical converter.properties file appears below:

```
Shift_JIS=CP943C
EUC-JP=Cp33722C
EUC-JP=Cp33722C
EUC-KR=Cp970
EUC-TW=Cp964
Big5=Co950
GB2312=Cp1386
ISO-2022–KR=ISO2022KR
```

The converter.properties file implements a method for specifying a content type header field that browsers would understand (such as, SHIFT_JIS) and a writer that can output characters correctly (such as, Cp943c).

Language versions offered by this product

This product is offered in several languages, as enabled by the operating platform on which the product is installed.

For the z/OS platform, the following language versions are available:

- English
- Japanese

Globalization: Resources for learning

Use links in this topic to find relevant supplemental information about globalization. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and IBM Redbooks® publications that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:

- “Programming instructions and examples” on page 531
- “Programming specifications” on page 531

Programming instructions and examples

- Java internationalization tutorial

  An online tutorial that explains how to use the Java SDK Internationalization API.
- Globalize your On Demand Business

  IBM's portal site for delivering globalized applications.

Programming specifications

- Java 2 Platform Standard Edition 5.0 Development Kit Documentation: Internationalization

  The Java internationalization documentation from Sun Microsystems, including a list of supported locales and encodings. For other versions of the Java platform, click the “Internationalization Home Page” link on that page.
- Java Specification Request 150, Internationalization Service for J2EE

  The specification of the Java internationalization service that was developed through the Java Community Process.
- W3C, Internationalization Core Working Group

  The W3C's Internationalization Core Working Group responsible for investigating the internationalization of web services, in particular, the dependence of web services on language, culture, region, and locale-related contexts.
- Making the WWW truly World Wide

  The W3C effort to make web technologies work with the many writing systems, languages, and cultural conventions of the global community:

Task overview: Internationalizing interface strings (localizable-text API)

This topic summarizes the steps involved in implementing message catalogs through the localizable-text API.

About this task

This product supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

Procedure

1. Identify localizable text in your application.
2. Create the message catalogs that are necessary for the locales to be supported by your application.
3. In your application code, compose the language-specific strings for output.
4. Using an assembly tool, assemble your application code as one or more application components.
5. Prepare the localizable-text package for deployment with your localized application. In this step, you create a deployment Java archive (JAR) file.
6. Assemble the application modules and the deployment JAR file into a Java Platform, Enterprise Edition (Java EE) application.
7. Deploy and manage the application.

Results

Your application is deployed with localized text.

Identifying localizable text

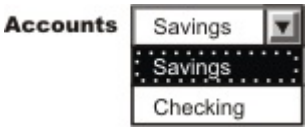
The first step in localizing strings in an application component is identifying the best candidates for translation.

Procedure

1. Determine which elements of the application need translating. Good candidates for localization include the following:
   - Graphical user interfaces: window titles, menus and menu items, buttons, on-screen instructions
   - Prompts in command-line interfaces
   - Application output: messages and logs
2. Assign a unique key to each element for use in message catalogs for the application. The key provides a language-neutral link between the application and language-specific strings in the message catalogs. Establishing a naming convention for keys before creating the catalogs can make writing code with these keys much more intuitive for interface programmers.

Example

Suppose you are localizing the GUI for a banking system, and the first window contains a pull-down list to use for selecting a type of account.



The labels for the list and the account types in the list are good choices for localization. Three elements require keys: the list and two items in the list.

What to do next

Create message catalogs for the language-specific strings.

Creating message catalogs

Perform this task to begin the localization of strings in an application component.

Before you begin

Identify strings that need to be localized.

About this task

You can create a catalog as either a java.util.ResourceBundle subclass or a Java properties file. The properties-file approach is more common, because properties files can be prepared by people without programming experience and swapped without modifying the application code.

Procedure

1. For each string that is identified for localization, add a line to the message catalog that lists the string key and value in the current language. In a properties file, each line has the following structure:

   ```
   key = string associated with the key
   ```

2. Save the catalog, giving it a locale-specific name. To enable resolution to a specific properties file, the Java API specifies naming conventions for the properties files in a resource bundle as *bundleName_localeID*.properties. Give the set of message catalogs a collective name, for example, BankingResources. For information about locale IDs that are recognized by the Java APIs, see “Resources for learning”.

Example

The following English catalog (BankingResources_en.properties) supports the labels for the list and its two list items:

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
```

Do not create compound strings by concatenation (for example, combining the values of savingsString and accountString to form `Savings Accounts` in English. Success depends upon the grammar of the original language (in this case, English) and is not likely to extend to other languages.

The corresponding German catalog (BankingResources_de.properties) supports the labels as follows:

```
accountString = Konten
savingsString = Sparkonto
checkingString = Girokonto
```

What to do next

Write code to compose the language-specific strings.

Composing language-specific strings

Perform this task to complete the localization of strings in an application component.

Before you begin

Create message catalogs for the language-specific strings.

Procedure

1. In application code, create a LocalizableTextFormatter instance, passing in required localization values.
2. Set other localization values as needed for more complex situations.
3. Generate a properly formatted, language-specific string.

What to do next

When the application is finished, deploy your application. For more information, see “Preparing the localizable-text package for deployment” on page 541.

Localization API support

The com.ibm.websphere.i18n.localizabletext package contains classes and interfaces for localizing text.

This package makes extensive use of the internationalization features of the standard Java APIs, including the following classes:

- java.util.Locale
- java.util.TimeZone
- java.util.ResourceBundle
- java.text.MessageFormat

For more information about the standard Java APIs, see “Globalization: Resources for learning” on page 530.

The localizable-text package wraps the Java support and extends it for efficient and simple use in a distributed environment. The primary class used by application programmers is LocalizableTextFormatter. Instances of this class are usually created in server programs, but client programs can also create them. Formatter instances are created for specific resource-bundle names and keys. Client programs that

receive a LocalizableTextFormatter instance call its format method. This method uses the locale of the client application to retrieve the appropriate resource bundle and compose a locale-specific message based on the key.

For example, suppose that a distributed application supports both French and English locales; the server is using an English locale and the client, a French locale. The server creates two resource bundles, one each for English and French. When the client makes a request that triggers a message, the server creates a LocalizableTextFormatter instance that contains the name of the resource bundle and the key for the message and passes the instance back to the client.

When the client receives the LocalizableTextFormatter instance, it calls the format method of the object. By using the locale and name of the resource bundle, the format method determines the name of the resource bundle that supports the French locale and retrieves the message that corresponds to the key from the French resource bundle. Formatting of the message is transparent to the client.

In this simple example, the resource bundles reside centrally with the server. They do not have to exist with the client. Part of what the localizable-text package provides is the infrastructure to support centralized catalogs. This implementation uses an enterprise bean (a stateless session bean provided with the localizable-text package) to access the message catalogs. When the client calls the format method on the LocalizableTextFormatter instance, the following events occur:

1. The client application sets the time-zone and locale values in the LocalizableTextFormatter instance, either by passing them explicitly or through default values.
2. A LocalizableTextFormatterEJBFinder call is made to retrieve a reference to the formatter bean.
3. Information from the LocalizableTextFormatter instance, including the time zone and locale of the client, is sent to the formatting bean.
4. The formatting bean uses the name of the resource bundle, the message key, the time zone, and the locale to compose a language-specific message.
5. The formatter bean returns the formatted message to the client.
6. The formatted message is inserted into the LocalizableTextFormatter instance and returned by the format method.

A call to the format method requires at most one remote call, to contact the formatter bean. As an alternative, the LocalizableTextFormatter instance can cache formatted messages, eliminating the remote call for subsequent uses. In addition, you can set a fallback string so that the application can return a readable string even if it cannot access the appropriate message catalog.

The resource bundles can be stored locally. The localizable-text package provides a static variable that indicates whether the bundles are stored locally (LocalizableConfiguration.LOCAL) or remotely (LocalizableConfiguration.REMOTE). However, the setting of this variable applies to all applications running within the same Java virtual machine.

LocalizableTextFormatter class

The LocalizableTextFormatter class, found in the com.ibm.websphere.i18n.localizabletext package, is the primary programming interface for using the localizable-text package. Instances of this class contain the information needed to create language-specific strings from keys and resource bundles.

The LocalizableTextFormatter class extends the java.lang.Object class and implements the following interfaces:

- java.io.Serializable
- com.ibm.websphere.i18n.localizabletext.LocalizableText
- com.ibm.websphere.i18n.localizabletext.LocalizableTextL
- com.ibm.websphere.i18n.localizabletext.LocalizableTextTZ
- com.ibm.websphere.i18n.localizabletext.LocalizableTextLTZ

Creation and initialization of class instances

The LocalizableTextFormatter class supports the following constructors:
- LocalizableTextFormatter()
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)

The LocalizableTextFormatter instance must have certain values, such as a resource-bundle name, a key, and the name of the formatting application. If you do not pass these values in by using the second constructor listed previously, you can set them separately by making the following calls:
- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)

You can use a fourth method, setArguments(Object[] args), to set optional localization values after construction. See “Processing of application-specific values” on page 536 at the end of this topic. For a usage example, see “Composing complex strings” on page 538.

API for formatting text

The formatting methods in the LocalizableTextFormatter class generate a string from a set of message keys and resource bundles, based on some combination of locale and time-zone values. Each method corresponds to one of the four localizable-text interfaces implemented. The following list indicates the interface in which each formatting method is defined:
- LocalizableText.format()
- LocalizableTextL.format(java.util.Locale locale)
- LocalizableTextTZ.format(java.util.TimeZone timeZone)
- LocalizableTextLTZ.format(java.util.Locale locale, java.util.TimeZone timeZone)

The format method with no arguments uses the locale and time-zone values set as defaults for the Java virtual machine. All four methods issue LocalizableException objects as needed.

Location of message catalogs and the appName value

Applications written with the localizable-text package can access message catalogs locally or remotely. In a distributed environment, the use of remote, centrally located message catalogs is appropriate. All clients can use the same catalogs, and maintenance of the catalogs is simplified. Local formatting is useful in test situations and appropriate under some circumstances. To support either local or remote formatting, a LocalizableTextFormatter instance must indicate the name of the formatting application.

For example, when an application formats a message by using remote catalogs, the message is actually formatted by an enterprise bean on the server. Although the localizable-text package contains the code to automate the lookup of the formatter bean and to issue a call to it, the application needs to know the name of the formatter bean. Several methods in the LocalizableTextFormatter class use a value described as `appName`, which refers to the name of the formatting application. It is not necessarily the name of the application in which the value is set.

Caching of messages

LocalizableTextFormatter instances can optionally cache formatted messages so that they do not require reformatting when needed again. By default, caching is not enabled, but you can use a `LocalizableTextFormatter.setCacheSetting(true)` call to enable caching. When caching is enabled and the format method is called, the method determines whether the message is already formatted. If so, the cached message is returned. If the message is not found in the cache, the message is formatted and returned to the caller, and a copy of the message is cached for future use.

If caching is disabled after messages are cached, those messages remain in the cache until the cache is cleared by a call to the LocalizableTextformatter.clearCache method. You can clear the cache at any time; the cache is automatically cleared when any of the following methods is called:
- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)
- setArguments(Object[] args)

API for providing fallback information

Under some circumstances, it can be impossible to format a message. The localizable-text package implements a fallback strategy, making it possible to get some information even if a message cannot be formatted correctly into the requested language. The LocalizableTextFormatter instance can optionally store fallback values for a message string, the time zone, and the locale. These values can be ignored unless the LocalizableTextFormatter instance issues an exception. To set fallback values, call the following methods as appropriate:
- setFallBackString(String message)
- setFallBackLocale(Locale locale)
- setFallBackTimeZone(TimeZone timeZone)

For a usage example, see “Generating localized text” on page 540.

Processing of application-specific values

The localizable-text package provides native support for localization based on time zone and locale, but you can construct messages on the basis of other values as well. If you need to consider variables other than locale and time zone in formatting localized text, write your own formatter class.

Your formatter class can extend the LocalizableTextFormatter class or independently implement some or all of the same localizable-text interfaces. As a minimum, your class must implement the java.io.Serializable interface and at least one of the localizable-text interfaces and its corresponding format method. If your class implements more than one localizable-text interface and format method, the order of evaluation of the interfaces is as follows:
1. LocalizableTextLTZ
2. LocalizableTextL
3. LocalizableTextTZ
4. LocalizableText

As an example, the localizable-text package provides a class that reports the time and date (LocalizableTextDateTimeArgument). In that class, date and time formatting is localized in accordance with three values: locale, time zone, and style.

Creating a formatter instance

Perform this task to set localization values for strings in an application component.

About this task

Server programs typically create LocalizableTextFormatter instances that are sent to clients as the result of some operation; clients format the objects at the appropriate time. Less typically, client programs create LocalizableTextFormatter objects locally.

Procedure

1. If needed for your application, write your own formatter class. For more information about implementation, see “LocalizableTextFormatter class” on page 534.
2. In application code, call the appropriate constructor for the formatter class and set required localization values. Some localization values, such as resource bundle name, key and formatting application, must

be set, either through a constructor or soon after construction. Other localization values can be set only as needed. For more information about the API, see the related reference.

Example

The following code creates a LocalizableTextFormatter instance by using the default constructor and then sets the required localization values:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType) {
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ltf.setPatternKey("accountNumber");
   ltf.setResourceBundleName("BankingSample.BankingResources");
   ltf.setApplicationName("BankingSample");
   ...
}
```

The line of code in boldface exploits default behavior of the Java platform. By default, the Java platform looks first for a subclass of java.util.ResourceBundle called BankingResources. When none is found, the Java platform looks for a valid properties file of the same name. In this continuing example, a properties file is found.

The application that is requesting a localized message can specify the locale and time zone for message formatting, or the application can use the default values set for the Java virtual machine.

For example, a GUI can enable users to select the language in which to display the interface. A default value must be set initially so that the GUI can be created properly when the application first starts, but users can then change the locale for the GUI to suit their needs. The following code shows how to change the locale used by an application based on the selection of a menu item:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
...
import java.util.Locale;

public void actionPerformed(ActionEvent event) {
   String action = event.getActionCommand();
   ...
   if (action.equals("en_us")) {
      applicationLocale = new Locale("en", "US");
      ...
   }
   if (action.equals("de_de")) {
      applicationLocale = new Locale("de", "DE");
      ...
   }
   if (action.equals("fr_fr")) {
      applicationLocale = new Locale("fr", "FR");
      ...
   }
   ...
}
```

For more information, see “Generating localized text”.

What to do next

Set optional localization values.

Setting optional localization values

In addition to setting localization values that are required by the LocalizableTextFormatter interface, you can set a number of optional values in application code, either through the constructor or by calling any of several methods for that purpose.

About this task

With optional values, you can do the following actions:
- Compose complex strings from variable substrings
- Customize the formatting of strings, considering variables other than time zone and locale

Procedure

1. In application code, add the optional values into an array of type Object.
   ```
   Object[] arg = {new String(getAccountNumber())};
   ```
2. Pass the array into a LocalizableTextFormatter instance. You can pass the array through the appropriate constructor or call the setArguments(Object[]) method. For a usage example, see "Composing complex strings."

   Because the array is passed by value rather than by reference, any updates to the array variable after this point are not reflected in the LocalizableTextFormatter instance unless it is reset by calling the setArguments(Object[]) method.

What to do next

Write code to generate the localized text.

Composing complex strings:

Perform this task to insert variable substrings into a localized string.

Before you begin

Identify strings that need to be localized.

About this task

The localized-text package supports the substitution of variable substrings into a localized string that is retrieved from the message catalog by key.

Procedure

1. In the message catalog, specify the location of the substitution in the string to be retrieved. Variable components are designated by braces (for example, `{0}`).
2. In application code, create a LocalizableTextFormatter instance, passing in an array that contains the variable value. If the variable substring must be localized, you can create a nested LocalizableTextFormatter instance and pass the instance in instead of a value.
3. Generate a localized string. When a format method is called on a formatter instance, the formatter takes each element of the array passed in the previous step and substitutes it for the placeholder with the matching index in the string that is retrieved from the message catalog. For example, the value at index 0 in the array replaces the `{0}` variable in the retrieved string.

Example

The following line from an English message catalog shows a string with a single substitution:

```
successfulTransaction = The operation on account {0} was successful.
```

The same key in message catalogs for other languages has a translation of this string with the variable at the appropriate location for each language.

The following code shows the creation of a single-element argument array and the creation and use of a LocalizableTextFormatter instance:

```
public void updateAccount(String transactionType) {
   ...
   Object[] arg = {new String(this.accountNumber)};
   ...
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter ("BankingResources",
                                    "successfulTransaction",
                                    "BankingSample",
                                    arg);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

Nesting formatter instances for localized substrings:

The ability to substitute variable substrings into the strings retrieved from message catalogs adds a level of flexibility to the localizable-text package, but this capability is of limited use unless the variable value can be localized. You can localize this value by nesting LocalizableTextFormatter instances.

Before you begin

Identify strings that need to be localized.

Procedure

1. In the message catalog, add entries that correspond to potential values for the variable substring.
2. In application code, create a LocalizableTextFormatter instance for the variable substring, setting required localization values.
3. Create a LocalizableTextFormatter instance for the primary string, passing in an array that contains the formatter instance for the variable substring.

Example

The following line from an English message catalog shows a string entry with two substitutions and entries to support the localizable variable at index 0 (the second variable in the string, the account number, does not need to be localized):

```
successfulTransaction = The {0} operation on account {1} was successful.
depositOpString = deposit
withdrawOpString = withdrawal
```

The following code shows the creation of the nested formatter instance and its insertion (with the account number variable) into the primary formatter instance:

```
public void updateAccount(String transactionType) {
   ...
   // Successful deposit
   LocalizableTextFormatter opLTF =
      new LocalizableTextFormatter("BankingResources",
                                   "depositOpString",
               "BankingSample");
   Object[] args = {opLTF, new String(this.accountNumber)};
   ...
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter ("BankingResources",
                                    "successfulTransaction",
```

```
                                    "BankingSample",
                                    args);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

Generating localized text

Perform this task to specify the runtime formatting of localized text in an application component.

Before you begin

Create a formatter instance and set the localization values as needed.

Procedure

1. If needed, customize the formatting behavior.
2. In application code, call the appropriate format method.

Example

You can provide fallback behavior for use if the appropriate message catalog is not available at formatting time.

The following code generates a localized string. If the formatting fails, the application retrieves and uses a fallback string instead of the localized string:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType){
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ...
   ltf.setFallBackString("Enter account number: ");
   try {
      msg = new Label(ltf.format(this.applicationLocale), Label.CENTER);
   }
   catch (LocalizableException le) {
      msg = new Label(ltf.getFallBackString(), Label.CENTER);
   }
   ...
}
```

What to do next

When the application is finished, deploy your application. For more information, see “Preparing the localizable-text package for deployment” on page 541.

Customizing the behavior of a formatting method:

Perform this task to change the runtime formatting of localized strings in an application component.

About this task

You can customize formatting behavior by passing your own formatter classes into a LocalizableTextFormatter instance through an array of optional values. This action enables you to consider variables other than locale and time zone when formatting localized text.

Procedure

1. Write your own formatter class. For more information about implementation, see "LocalizableTextFormatter class."
2. In application code, create an instance of your formatter class as appropriate and pass it with any other optional localization values into an instance of LocalizableTextFormatter. When the LocalizableTextFormatter instance reads the instance that has been passed in, it attempts to call the format() method on the passed-in instance. The string returned is then processed with any other elements in the array.

Example

The localizable-text package provides an example of a user-defined class, called LocalizableTextDateTimeArgument. This class enables date and time information to be selectively formatted according to the style values defined in the java.text.DateFormat interface as well as the constants that are defined within the LocalizableTextDateTimeArgument class.

Preparing the localizable-text package for deployment

The LocalizableTextEJBDeploy tool is used to create a deployment Java Archive (JAR) file for the localizable text service. You must deploy the enterprise bean in each enterprise application that requires support for localized text.

Before you begin

Write code to compose the language-specific strings.

Procedure

1. Make sure that the LocalizableTextEJBDeploy tool is included in the class path.

   transition: In versions 6.0.x and earlier, the LocalizableTextEJBDeploy tool used to reside in the file *app_server_root*/`lib/ltext.jar`. It now resides in the file *app_server_root*/`plugins/com.ibm.ws.runtime_1.0.0.jar`.
2. Set up a working directory for the LocalizableTextEJBDeploy tool to use. You need to pass this location to the tool through a command-line interface.
3. Run the LocalizableTextEJBDeploy tool. You might be asked if you want to regenerate deployment code for the LocalizableText bean. Do not redeploy the bean; if you do, an incorrect Java Naming and Directory Interface (JNDI) name will be generated.

   To deploy the bean on multiple hosts and servers, run the tool for each host and server combination. This action generates a unique JNDI name for each deployment. After the tool is run, a deployment JAR file is located in the working directory that you specified.

What to do next

Using an assembly tool, assemble the deployment JAR file in an enterprise application with other application components.

As part of preparing for deployment, perform the following:
- Add the resource bundles for your application to the Enterprise Archive (EAR) file as files.
- Add the location of the EAR file to the server class path for the server so that the resource bundles can be located on the virtual host and server.

The same deployment JAR file can be included in several enterprise applications.

LocalizableTextEJBDeploy command

This topic describes the command-line syntax for the LocalizableTextEJBDeploy tool.

transition: In versions 6.0.x and earlier, the LocalizableTextEJBDeploy tool used to reside in the file *app_server_root*/lib/ltext.jar. It now resides in the file *app_server_root*/plugins/com.ibm.ws.runtime_1.0.0.jar.

```
LocalizableTextEJBDeploy
   -a applicationName
   -h virtualHostName
   -i installationDirectory
   -s serverName
   -w workingDirectory
```

Parameters

The required parameters, which can be specified in any order, follow:

applicationName
The name of the formatting session bean. This name is used in LocalizableTextFormatter instances to specify where the actual formatting occurs. If the name cannot be resolved at run time, the format method issues an exception.

virtualHostName
The name of the virtual host on which the formatting session bean is deployed. This value is case-sensitive on all operating platforms.

installationDirectory
The location at which the application server product is installed.

serverName
The name of the application server. If this argument is not specified, the default server name for the product is used.

workingDirectory
A location for the tool to use temporarily.

Task overview: Internationalizing application components (internationalization service)

This topic summarizes the steps involved in using the internationalization service.

About this task

With the internationalization service, you can manage the distribution of the internationalization information, or *internationalization context*, that is necessary to support globalized Java Platform, Enterprise Edition (Java EE) application components. Supported application components also include web service client environments and web service-enabled enterprise beans.

Procedure

1. Use the internationalization context API within application components to obtain or manage internationalization context.

   Servlet and enterprise bean business methods can use internationalization context to perform locale- and time zone-sensitive localizations. Enterprise JavaBeans (EJB) client applications, and server components that are configured to manage internationalization context must use the internationalization context API to set the context elements scoped to their invocations.

   You use the internationalization context API within Web service-enabled Java EE client programs and stateless session beans in the same manner that you would use conventional Java EE application components, with one exception. Internationalization context propagated over Web service requests contains a time zone ID, whereas conventional Remote Method Invocation/ Internet Inter-ORB Protocol (RMI/IIOP) requests propagate complete time zone information, including the raw offset, Daylight Savings Time information, and so on.

2. Assemble internationalized applications.

   The internationalization type specifies the internationalization policy that applies to a servlet or an enterprise bean and, in particular, indicates whether the application component or its hosting Java EE container manages internationalization context. Container internationalization attributes can be specified for container-managed servlet and enterprise bean business methods. These attributes tailor a policy by indicating which context the container scopes to an invocation. Configuring internationalization policies declaratively prescribes, by means of the application deployment descriptor, the distribution and management of context throughout an application.

   As you edit the deployment descriptor for assembly, you can also set the internationalization type and configure any container internationalization attributes for the servlets and enterprise beans in your application.

   You configure internationalization type and container internationalization attributes for Web service-enabled stateless session beans in the same manner as you do for conventional beans.
3. Manage the internationalization service.

   Use the administrative console to enable the service on all application servers.

   By default, the service is enabled within Java EE client environments but is disabled on application servers. You must enable the service on all application servers hosting your servlets and enterprise beans to use internationalization context.
4. Troubleshoot the internationalization service as needed.

   Use the administrative console to enable the trace service to log internationalization service messages when debugging your applications.

   The trace strings for the internationalization service follow; use both:

   ```
   com.ibm.ws.i18n.context.*=all=enabled:com.ibm.websphere.i18n.context.*=all=enabled
   ```

Internationalization service

In a distributed client-server environment, application processes can run on different machines, configured for different locales, corresponding to different cultural conventions; they can also be located across geographical boundaries. The internationalization service can help manage your application in a globally distributed environment.

For an understanding of how differences in locale impact application development, read “Globalization” on page 527.

Java Platform, Enterprise Edition (Java EE) provides support for application components that run on computers with differing endian architecture and code sets. It does not provide dedicated support for application components that run on computers with different locales or time zones.

The internationalization service addresses the challenges posed by locale and time zone mismatch without incurring the limitations of conventional techniques. The service systematically manages the distribution of internationalization contexts across the various components of EJB applications, including client applications, enterprise beans, and servlets.

The service works by associating an internationalization context with every service request within an application. When a client-side component calls a business method, the internationalization service interposes by obtaining the internationalization context associated with the current client-side process and by attaching that context to the outgoing request. On the server side, the internationalization service again interposes by detaching the context from the incoming request and associating it with the server-side process on which the business method will run, effectively scoping the context to the business method. For HTTP requests, the caller context is constructed from the HTTP attributes and default values. The service propagates internationalization context on subsequent business method invocations in the same manner, which distributes the context of the originating request over the entire chain of business method invocations.

This basic operation of scoping and propagation is defined precisely by *internationalization context management policies*. Internationalization policies specify whether an application component or its hosting Java EE container are to manage internationalization context. For container-managed components, the policy indicates which internationalization context the container scopes to invocations on that component. Server components configured to manage internationalization context, as well as EJB clients, must use the internationalization context API to manage the internationalization context elements scoped to their invocations.

Every application component has a default policy, which can be overridden and tailored for servlets and enterprise beans at assembly time.

At run time, application components can use the internationalization context API to get any element of the internationalization contexts scoped to an invocation. To programmatically access context elements, application components first resolve an internationalization context API reference, then call the appropriate API method to access the various context elements, such as the caller locale or the invocation time zone. These elements can be used in calls to Java SDK internationalization API methods; for example, to perform localizations such as formatting messages, configuring dates, or comparing strings.

Assembling internationalized applications

Perform this task to configure application components for deployment with the internationalization service.

About this task

Use an assembly tool to configure internationalization in the deployment descriptors for servlets and enterprise beans.

Procedure

1. Set the **internationalization type**.

   All servlets and enterprise beans have an internationalization type setting that specifies whether internationalization context is managed by the application component or by its hosting Java Platform, Enterprise Edition (Java EE) container during invocations of their respective life cycle and business methods. The internationalization type can be configured for all server application components except entity beans, which are container-managed only.

   By default, all server components use container-managed internationalization (CMI). The default setting suffices in most cases; when it does not, modify the internationalization type setting by completing the steps that are described in one of the following topics:
   - “Setting the internationalization type for servlets”
   - “Setting the internationalization type for enterprise beans” on page 546
2. Set the **container internationalization attribute**.

   You can associate CMI servlets, and business methods of CMI enterprise beans, with a container internationalization attribute. That attribute specifies which of three internationalization contexts (**Caller**, **Server**, or **Specified**) the container is to scope to an invocation. When running as specified, the container internationalization attribute also specifies the custom internationalization context elements.

   Named container internationalization attributes can be associated with sets of servlets or with sets of Enterprise JavaBeans (EJB) business methods. Initially, CMI servlets and business methods implicitly run as caller and do not associate with a container internationalization attribute. When the implicit behavior or an associated attribute setting is unsuitable, configure an attribute by completing the steps that are described in one of the following topics:
   - “Configuring container internationalization for servlets” on page 545
   - “Configuring container internationalization for enterprise beans” on page 547

Setting the internationalization type for servlets

This task sets the internationalization type for a servlet within a Web module.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported a dynamic Web project.

Procedure

1. In the Java EE perspective, open the Web project for which you want to set the internationalization type.
   a. Double-click **Dynamic Web Projects**.
   b. Double-click the name of the Web project to see its contents.
   c. Double-click the deployment descriptor object.

   The Web Deployment Descriptor panel is displayed.
2. In the Web Deployment Descriptor panel, click the Servlets tab.
3. Scroll down to **WebSphere Programming Model Extensions** and then **Internationalization**.
4. From the **Servlets and JSPs** list of the Servlets panel, select the servlet for which you want to set the internationalization type.
5. Under **Internationalization**, select a value from the **Internationalization type** list. Valid values are `Application` or `Container`.
6. From the menu bar, click **File > Save**.

Results

The internationalization type setting is assigned to the servlet.

What to do next

If you selected container-managed internationalization, you can then set container-managed internationalization attributes for methods within the servlet. For more information, see "Configuring container internationalization for servlets."

Configuring container internationalization for servlets

This task configures container internationalization for a servlet within a Web module.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported a dynamic Web project.

You must also have set the internationalization type of one or more servlets in a Web project to `Container`.

About this task

This procedure relates one or more servlets to a container-managed internationalization attribute.

Procedure

1. In the Java EE perspective, open the Web project for which you want to configure container internationalization.
   a. Double-click **Dynamic Web Projects**.
   b. Double-click the name of the Web project to see its contents.
   c. Double-click the deployment descriptor object.

   The Web Deployment Descriptor panel is displayed.
2. In the Web Deployment Descriptor panel, click the Servlets tab.
3. Scroll down to **WebSphere Programming Model Extensions** and then **Internationalization**.
4. Following **Container-managed Internationalization Attribute**, set the **Run As** field by selecting `Caller`, `Server`, or `Specified`.
5. If the servlet is to be run as Specified, select an internationalization policy from the **Specified** list or define a new policy.
   a. To define an internationalization policy, click **New**. The New Specified Initialization wizard is displayed.
   b. In the **Description** field, give the policy a name.
   c. If needed, set a time zone ID and add a time zone description. If you do not find the appropriate time zone in the ID list, click **Customize** to define one relative to Greenwich Mean Time (GMT).
   d. Create at least one locale for the policy. To create a locale, click **Add**; select a language and (optional) geographic region; specify a variant as needed. Add a locale description and click **OK** to finish. The new locale is added to the **Locales** list.
   e. If more than one locale is defined for the policy, select a locale from the **Locales** list and click **Finish**. Otherwise, just click **Finish**
6. From the menu bar, click **File > Save**.

Results

Selected servlets are now configured to run under the associated internationalization settings.

Setting the internationalization type for enterprise beans

This task sets the internationalization type for an enterprise bean within an Enterprise JavaBeans (EJB) module.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported an EJB project.

About this task

Container-managed internationalization (CMI) is the default type; entity beans cannot be set to application-managed internationalization (AMI). Use CMI also for stateless session beans that are enabled for Web services.

Procedure

1. In the Java EE perspective, open the EJB project for which you want to set the internationalization type.

a. Double-click **EJB Projects**.
b. Double-click the name of the EJB project to see its contents.
c. Double-click the deployment descriptor object.

The EJB Deployment Descriptor panel is displayed.

2. In the EJB Deployment Descriptor panel, click the Internationalization tab. Any enterprise beans that are already configured for AMI are displayed in the **Internationalization type** list.
3. To set the internationalization type for any other enterprise beans to AMI, click **Add** following the **Internationalization type** list. The Internationalization Type wizard opens. Only message-driven or session beans can be selected.
4. Select the beans that you want to set and click **Finish** to exit the wizard.
5. From the menu bar, click **File > Save**.

Results

The internationalization type is assigned to the bean.

What to do next

For beans that use container-managed internationalization, you can then set container-managed internationalization attributes. For more information, see "Configuring container internationalization for enterprise beans."

Configuring container internationalization for enterprise beans

This task configures container internationalization for enterprise bean business methods.

Before you begin

This topic assumes that you have an assembly tool such as Rational Application Developer.

For information about assembly, refer to the documentation for your assembly tool. The steps in this topic refer to Rational Application Developer.

This topic assumes that you have started the assembly tool, configured the assembly tool for work on Java Platform, Enterprise Edition (Java EE) modules, and created or imported an EJB project.

You must also have one or more enterprise beans set to container-managed internationalization (CMI) by default.

About this task

This procedure relates one or more business methods to one or more container-managed internationalization (CMI) attributes. Use this procedure also for stateless session beans that are enabled for Web services.

Procedure

1. In the Java EE perspective, open the EJB project for which you want to configure container internationalization.
   a. Double-click **EJB Projects**.
   b. Double-click the name of the EJB project to see its contents.
   c. Double-click the deployment descriptor object.

   The EJB Deployment Descriptor panel is displayed.

2. In the EJB Deployment Descriptor panel, click the Internationalization tab. Any business methods that are already configured are displayed in the **Internationalization attributes** list.
3. To configure a CMI business method, click **Add** following the **Internationalization attributes** list. The Internationalization Attributes wizard opens.
4. Set the **Run As** field by selecting `Caller`, `Server`, or `Specified`. Add a meaningful description. As a group, the CMI attribute settings comprise an internationalization policy.
   - The description appears as `Internationalization` *`description`* (*`runAsSetting`*) in the **Internationalization attributes** list when you are finished.
   - If you do not provide a description, the policy name appears as `Internationalization` (*`runAsSetting`*).

   If the bean is to be run as Specified, complete the following steps to specify the context elements that the container scopes to bean method invocations.
   a. Set a time zone ID and add a time zone description as needed. If you do not find the appropriate time zone in the ID list, click **Custom** to define one relative to Greenwich Mean Time (GMT).
   b. Set a locale. Select a language and (optional) geographic region; specify a variant as needed. Add a locale description as needed and click **OK** to finish.
5. Click **Next**.
6. Select the beans for which you want to configure method-level internationalization attributes and click **Next**.
7. Select the methods that you want to configure and click **Next**. A check box is displayed next to each method name that you select.
   - Click **Apply to All** to place a check box next to the displayed bean name.
   - Click **Select Beans** to select more beans with CMI.
8. Click **Finish** to exit the wizard.
9. From the menu bar, click **File > Save**.

Results

The bean methods are now configured to run under the associated internationalization settings.

Using the internationalization context API

Enterprise JavaBeans (EJB) client applications, servlets, and enterprise beans can programmatically obtain and manage internationalization context using the internationalization context API. For Web service client applications, you use the API to obtain and manage internationalization context in the same manner as for EJB clients.

Before you begin

The java.util and com.ibm.websphere.i18n.context packages contain all of the classes necessary to use the internationalization service within an EJB application.

Procedure

1. Gain access to the internationalization context API.

   Resolve internationalization context API references once over the life cycle of an application component, within the initialization method of that component (for example, within the init method of servlets, or within the SetXxxContext method of enterprise beans). For Web service client programs, resolve a reference to the internationalization context API during initialization. For stateless session beans enabled for Web services, resolve the reference in the setSessionContext method.
2. Access caller locales and time zones.

   Every remote invocation of an application component has an associated caller internationalization context associated with the thread that is running that invocation. A caller context is propagated by the

internationalization service and middleware to the target of a request, such as an Enterprise JavaBeans (EJB) business method or servlet service method. This task also applies to Web service client programs.

3. Access invocation locales and time zones.

   Every remote invocation of a servlet service or Enterprise JavaBeans (EJB) business method has an invocation internationalization context associated with the thread that is running that invocation. Invocation context is the internationalization context under which servlet and business method implementations run; it is propagated on subsequent invocations by the internationalization service and middleware. This task also applies to Web service client programs.

Results

The resulting components are said to use *application-managed internationalization* (AMI). For more information about AMI, see “Internationalization context: Management policies” on page 564.

Example

Each supported application component uses the internationalization context API differently. Code examples are provided that illustrate how to use the API within each component type. Differences in API usage, as well as other coding tips, are noted in comments that precede the relevant statement blocks.

- Managing internationalization context in an EJB client program
- Managing internationalization context in a servlet
- Managing internationalization context in a session bean
- Representing internationalization context in a SOAP header

Managing internationalization context in an EJB client program: The following code example illustrates how to use the internationalization context API within a contained EJB client program or Web service client program.

```
//----------------------------------------
// Basic Example: J2EE EJB client.
//----------------------------------------
package examples.basic;

//----------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//----------------------------------------
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;
import java.util.SimpleTimeZone;

public class EjbClient {

  public static void main(String args[]) {

    //------------------------------------------------
    // INTERNATIONALIZATION SERVICE: API references.
    //------------------------------------------------
    UserInternationalization userI18n = null;
    Internationalization callerI18n = null;
    InvocationInternationalization invocationI18n = null;

    //------------------------------------------------
    // INTERNATIONALIZATION SERVICE: JNDI name.
```

```
  //-------------------------------------------------
  final String UserI18NUrl =
      "java:comp/websphere/UserInternationalization";

  //-------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Resolve the API.
  //-------------------------------------------------
  try {
   Context initialContext = new InitialContext();
   userI18n = (UserInternationalization)initialContext.lookup(
       UserI18NUrl);
   callerI18n = userI18n.getCallerInternationalization();
   invI18n = userI18n.getInvocationInternationalization ();
  } catch (NamingException ne) {
    log("Error: Cannot resolve UserInternationalization: Exception: " + ne);
  } catch (IllegalStateException ise) {
    log("Error: UserInternationalization is not available: " + ise);
  }
  ...

  //-------------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Set invocation context.
  //
  // Under Application-managed Internationalization (AMI), contained EJB
  // client programs may set invocation context elements. The following
  // statements associate the supplied invocation locale and time zone
  // with the current thread. Subsequent remote bean method calls will
  // propagate these context elements.
  //-------------------------------------------------------------------
  try {
    invocationI18n.setLocale(new Locale("fr", "FR", ""));
    invocationI18n.setTimeZone("ECT");
  } catch (IllegalStateException ise) {
    log("An anomaly occurred accessing Invocation context: " + ise );
  }
  ...

  //-------------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Get locale and time zone.
  //
  // Under AMI, contained EJB client programs can get caller and
  // invocation context elements associated with the current thread.
  // The next four statements return the invocation locale and time zone
  // associated above, and the caller locale and time zone associated
  // internally by the service. Getting a caller context element within
  // a contained client results in the default element of the JVM.
  //-------------------------------------------------------------------
  Locale invocationLocale = null;
  SimpleTimeZone invocationTimeZone = null;
  Locale callerLocale = null;
  SimpleTimeZone callerTimeZone = null;
  try {
    invocationLocale = invocationI18n.getLocale();
    invocationTimeZone =
        (SimpleTimeZone)invocationI18n.getTimeZone();
    callerLocale = callerI18n.getLocale();
    callerTimeZone = SimpleTimeZone)callerI18n.getTimeZone();
  } catch (IllegalStateException ise) {
    log("An anomaly occurred accessing I18n context: " + ise );
  }

  ...
} // main

...
```

```
  void log(String s) {
    System.out.println (((s == null) ? "null" : s));
  }
} // EjbClient
```

Managing internationalization context in a servlet: The following code example illustrates how to use the internationalization context API within a servlet. Note comments in the init and doPost methods.

```
...
//------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//------------------------------------------------------------------
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;

public class J2eeServlet extends HttpServlet {

  ...
  //------------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: API references.
  //------------------------------------------------------------------
  protected UserInternationalization userI18n = null;
  protected Internationalization i18n = null;
  protected InvocationInternationalization invI18n = null;

  //------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: JNDI name.
  //------------------------------------------------------------
  public static final String UserI18NUrl =
      "java:comp/websphere/UserInternationalization";

  protected Locale callerLocale = null;
  protected Locale invocationLocale = null;

  /**
   * Initialize this servlet.
   * Resolve references to the JNDI initial context and the
   * internationalization context API.
   */
  public void init() throws ServletException {

    //------------------------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Resolve API.
    //
    // Under Container-managed Internationalization (CMI), servlets have
    // read-only access to invocation context elements. Attempts to set these
    // elements result in an IllegalStateException.
    //
    // Suggestion: cache all internationalization context API references
    // once, during initialization, and use them throughout the servlet
    // lifecycle.
    //------------------------------------------------------------------
    try {
      Context initialContext = new InitialContext();
      userI18n = (UserInternationalization)initialContext.lookup(UserI18nUrl);
      callerI18n = userI18n.getCallerInternationalization();
      invI18n = userI18n.getInvocationInternationalization();
    } catch (NamingException ne) {
      throw new ServletException("Cannot resolve UserInternationalization" + ne);
    } catch (IllegalStateException ise) {
      throw new ServletException ("Error: UserInternationalization is not
```

```
        available: " + ise);
    }
    ...
} // init

/**
 * Process incoming HTTP GET requests.
 * @param request Object that encapsulates the request to the servlet
 * @param response Object that encapsulates the response from the
 *    Servlet.
 */
public void doGet(
    HttpServletRequest  request,
    HttpServletResponse response)
  throws ServletException, IOException {
  doPost(request, response);
} // doGet

/**
 * Process incoming HTTP POST requests
 * @param request Object that encapsulates the request to
 *    the Servlet.
 * @param response Object that encapsulates the response from
 *    the Servlet.
 */
public void doPost(
    HttpServletRequest  request,
    HttpServletResponse response)
  throws ServletException, IOException {

  ...
  //---------------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: Get caller context.
  //
  // The internationalization service extracts the accept-languages
  // propagated in the HTTP request and associates them with the
  // current thread as a list of locales within the caller context.
  // These locales are accessible within HTTP Servlet service methods
  // using the caller internationalization object.
  //
  // If the incoming HTTP request does not contain accept languages,
  // the service associates the server's default locale. The service
  // always associates the GMT time zone.
  //
  //---------------------------------------------------------------------
  try {
    callerLocale = callerI18n.getLocale();   // caller locale
    // the following code enables you to get invocation locale,
    // which depends on the Internationalization policies.
    invocationLocale = invI18n.getLocale();  // invocation locale
  } catch (IllegalStateException ise) {
    log("An anomaly occurred accessing Invocation context: " + ise);
  }
  // NOTE: Browsers may propagate accept-languages that contain a
  // language code, but lack a country code, like "fr" to indicate
  // "French as spoken in France."  The following code supplies a
  // default country code in such cases.
  if (callerLocale.getCountry().equals(""))
    callerLocale = AccInfoJBean.getCompleteLocale(callerLocale);

  // Use iLocale in JDK locale-sensitive operations, etc.
  ...
} // doPost

...
```

```
  void log(String s) {
    System.out.println (((s == null) ? "null" : s));
  }
} // CLASS J2eeServlet
```

Managing internationalization context in a session bean: This code example illustrates how to perform a localized operation using the internationalization service within a session bean or Web service-enabled session bean.

```
...
//-------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//-------------------------------------------------------------
import com.ibm.websphere.i18n.context.UserInternationalization;
import com.ibm.websphere.i18n.context.Internationalization;
import com.ibm.websphere.i18n.context.InvocationInternationalization;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Locale;

/**
 * This is a stateless Session Bean Class
 */
public class J2EESessionBean implements SessionBean {

  //-------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: API references.
  //-------------------------------------------------------------
  protected UserInternationalization        userI18n = null;
  protected InvocationInternationalization  invI18n  = null;

  //-------------------------------------------------------------
  // INTERNATIONALIZATION SERVICE: JNDI name.
  //-------------------------------------------------------------
  public static final String UserI18NUrl =
      "java:comp/websphere/UserInternationalization";
  ...

  /**
   * Obtain the appropriate internationalization interface
   * reference in this method.
   * @param ctx javax.ejb.SessionContext
   */
  public void setSessionContext(javax.ejb.SessionContext ctx) {

    //-------------------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Resolve the API.
    //-------------------------------------------------------------
    try {
      Context initialContext = new InitialContext();
      userI18n = (UserInternationalization)initialContext.lookup(
          UserI18NUrl);
      invI18n = userI18n.getInvocationInternationalization();
    } catch (NamingException ne) {
      log("Error: Cannot resolve UserInternationalization: Exception: " + ne);

    } catch (IllegalStateException ise) {
      log("Error: UserInternationalization is not available: " + ise);
    }
  } // setSessionContext

 /**
  * Set up resource bundle using I18n Service
  */
  public void setResourceBundle()
```

```
  {
    Locale invLocale   = null;

    //-------------------------------------------------------------
    // INTERNATIONALIZATION SERVICE: Get invocation context.
    //-------------------------------------------------------------
    try {
      invLocale = invI18n.getLocale();
    } catch (IllegalStateException ise) {
      log ("An anomaly occurred while accessing Invocation context: " + ise );
    }
    try {
      Resources.setResourceBundle(invLocale);
      // Class Resources provides support for retrieving messages from
      // the resource bundle(s). See Currency Exchange sample source code.
    } catch (Exception e) {
      log("Error: Exception occurred while setting resource bundle: " + e);
    }
  } // setResourceBundle

  /**
   * Pass message keys to get the localized texts
   * @return java.lang.String []
   * @param key java.lang.String []
   */
  public String[] getMsgs(String[] key) {
    setResourceBundle();
    return Resources.getMsgs(key);
  }

  ...
  void log(String s) {
    System.out.println(((s == null) ? ";null" : s));
  }
} // CLASS J2EESessionBean
```

Representing internationalization context in a SOAP header: This code example illustrates how internationalization context is represented within the SOAP header of a Web service request.

```
<InternationalizationContext>
   <Locales>
      <Locale>
         <LanguageCode>ja</LanguageCode>
         <CountryCode>JP</CountryCode>
         <VariantCode>Nihonbushi</VariantCode>
      </Locale>
      <Locale>
         <LanguageCode>fr</LanguageCode>
         <CountryCode>FR</CountryCode>
      </Locale>
      <Locale>
         <LanguageCode>en</LanguageCode>
         <CountryCode>US</CountryCode>
      </Locale>
   </Locales>
   <TimeZoneID>JST</TimeZoneID>
</InternationalizationContext>
```

This representation is valid against the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="InternationalizationContext"
                 type="InternationalizationContextType">
    </xsd:element>

    <xsd:complexType name="InternationalizationContextType">
```

```
        <xsd:sequence>
            <xsd:element name="Locales"
                         type="LocalesType">
            </xsd:element>
            <xsd:element name="TimeZoneID"
                         type="xsd:string">
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LocalesType">
        <xsd:sequence>
            <xsd:element name="Locale"
                         type="LocaleType"
                         minOccurs="0"
                         maxOccurs="unbounded">
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="LocaleType">
        <xsd:sequence>
            <xsd:element name="LanguageCode"
                         type="xsd:string"
                         minOccurs="0"
                         maxOccurs="1">
            </xsd:element>
            <xsd:element name="CountryCode"
                         type="xsd:string"
                         minOccurs="0"
                         maxOccurs="1">
            </xsd:element>
            <xsd:element name="VariantCode"
                         type="xsd:string"
                         minOccurs="0"
                         maxOccurs="1">
            </xsd:element>
         </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Gaining access to the internationalization context API

Perform this task to access the internationalization service by resolving a reference to the internationalization context API.

About this task

Resolve internationalization context API references once over the life cycle of an application component, within the initialization method of that component (for example, within the init method of servlets, or within the SetXxxContext method of enterprise beans). For Web service client programs, resolve a reference to the internationalization context API during initialization. For stateless session beans enabled for Web services, resolve the reference in the setSessionContext method.

Procedure

1. Resolve a reference to the UserInternationalization interface by performing a lookup on the Java Naming and Directory Interface (JNDI) name `java:comp/websphere/UserInternationalization`. For example:

```
//---------------------------------------------------------------------
// Internationalization context imports.
//---------------------------------------------------------------------
import com.ibm.websphere.i18n.context.*;
import javax.naming.*;
```

```
...

public class MyApplication {
  ...

  //------------------------------------------------------------------
  // Resolve a reference to the UserInternationalization interface.
  //------------------------------------------------------------------
  InitialContext initCtx = null;
  UserInternationalization userI18n = null;
  final String UserI18nUrl = "java:comp/websphere/UserInternationalization";
  try {
    initCtx = new InitialContext();
    userI18n = (UserInternationalization)initCtx.lookup(UserI18nUrl);
  }
  catch (NamingException ne) {
    // UserInternationalization URL is unavailable.
  }
```

If the UserInternationalization object is unavailable because of an anomaly or a restriction, the JNDI lookup invocation issues a javax.naming.NameNotFoundException exception that contains the java.lang.IllegalStateException instance.

2. Use the UserInternationalization reference to create references to the CallerInternationalization or InvocationInternationalization objects, which provide access to elements of the Caller or Invocation internationalization contexts, respectively. The CallerInternationalization reference can be bound to the Internationalization interface only; the InvocationInternationalization reference can be bound to either the Internationalization or the InvocationInternationalization interfaces, depending on whether the application requires read-only or read-write access to the invocation context. For example:

```
  ...
  //------------------------------------------------------------------
  // Resolve references to the Internationalization and
  // InvocationInternationalization interfaces.
  //------------------------------------------------------------------
  Internationalization callerI18n = null;
  InvocationInternationalization invocationI18n = null;
  try {
    callerI18n = userI18n.getCallerInternationalization();
    invocationI18n = userI18n.getInvocationInternationalization();
  }
  catch (IllegalStateException ise) {
    // An Internationalization interface(s) is unavailable.
  }
```

Accessing caller locales and time zones

Perform this task to access elements of the caller internationalization context.

Before you begin

An application component must first resolve a reference to the CallerInternationalization object and then bind it to the Internationalization interface.

About this task

Every remote invocation of an application component has an associated caller internationalization context associated with the thread that is running that invocation. A caller context is propagated by the internationalization service and middleware to the target of a request, such as an Enterprise JavaBeans (EJB) business method or servlet service method. This task also applies to Web service client programs.

Procedure

1. Obtain the desired caller context elements.

```
java.util.Locale [] myLocales  = null;
try {
  myLocales  = callerI18n.getLocales();
}
catch (IllegalStateException ise) {
  // The Caller context is unavailable;
  // is the service started and enabled?
}
...
```

The Internationalization interface contains the following methods to get caller internationalization context elements:

- **Locale [] getLocales()** Returns the list of caller locales that are associated with the current thread.
- **Locale getLocale()** Returns the first in the list of caller locales that are associated with the current thread.
- **TimeZone getTimeZone()** Returns the SimpleTimeZone caller that is associated with the current thread.

The Internationalization interface supports read-only access to internationalization context within application components. Methods of the Internationalization interface are available to all EJB application components and are used in the same manner for each, but the method semantics vary according to the component type. For instance, when obtaining the caller locale within an EJB client application, the interface returns the default locale of the host Java virtual machine (JVM); in contrast, when obtaining caller context within a servlet service method (for example, doPost or doGet methods), the interface returns the first locale (accept-language) propagated within the corresponding HTML request. See Internationalization context for a discussion of how the service propagates internationalization context throughout an application.

2. Use the caller context elements to localize computations under a locale or time zone of the calling process.

```
DateFormat df = DateFormat.getDateInstance(myLocale);
String localizedDate = df.getDateInstance().format(aDateInstance);
...
```

Accessing invocation locales and time zones

Perform this task to access elements of the invocation internationalization context.

Before you begin

An application component must first resolve a reference to the InvocationInternationalization object and then bind it to the InvocationInternationalization interface of the internationalization context API.

About this task

Every remote invocation of a servlet service or Enterprise JavaBeans (EJB) business method has an invocation internationalization context associated with the thread that is running that invocation. Invocation context is the internationalization context under which servlet and business method implementations run; it is propagated on subsequent invocations by the internationalization service and middleware. This task also applies to Web service client programs.

Procedure

1. Obtain the desired invocation context elements.

```
java.util.Locale myLocale;
try {
  myLocale = invocationI18n.getLocale();
}
catch (IllegalStateException ise) {
  // The invocation context is unavailable;
  // is the service started and enabled?
}
...
```

The InvocationInternationalization interface contains the following methods to both get and set invocation internationalization context elements:

- **Locale [] getLocales()**. Returns the list of invocation locales that is associated with the current thread.
- **Locale getLocale()**. Returns the first in the list of invocation locales that is associated with the current thread.
- **TimeZone getTimeZone()**. Returns the SimpleTimeZone invocation that is associated with the current thread.
- **setLocales(Locale [])**. Sets the list of invocation locales that are associated with the current thread to the supplied list.
- **setLocale(Locale)**. Sets the list of invocation locales that are associated with the current thread to a list that contains the supplied locale.
- **setTimeZone(TimeZone)**. Sets the invocation time zone that is associated with the current thread to the supplied SimpleTimeZone.
- **setTimeZone(String)**. Sets the invocation time zone that is associated with the current thread to a SimpleTimeZone that has the supplied ID.

The InvocationInternationalization interface supports read and write access to invocation internationalization context within application components. However, according to internationalization context management policies, only components configured to manage internationalization context (application-managed internationalization, or AMI, components) have write access to invocation internationalization context elements. Calls to set invocation context elements within container-managed internationalization (CMI) application components result in a java.lang.IllegalStateException exception. Any differences in how application components can use InvocationInternationalization methods are explained in Internationalization context.

2. Use the invocation context elements to localize a computation under a locale or time zone of the calling process.

```
DateFormat df = DateFormat.getDateInstance(myLocale);
  String localizedDate = df.getDateInstance().format(aDateInstance);
  ...
```

Example

In the following code example, locale (en,GB) and simple time zone (GMT) transparently propagate on the call to the myBusinessMethod method. Server-side application components, such as `myEjb`, can use the InvocationInternationalization interface to obtain these context elements.

```
...
//--------------------------------------------------------------------
// Set the invocation context under which the business method or
// servlet will run and propagate on subsequent remote business
// method invocations.
//--------------------------------------------------------------------
try {
  invocationI18n.setLocale(new Locale("en", "GB"));
  invocationI18n.setTimeZone(SimpleTimeZone.getTimeZone("GMT"));
}
catch (IllegalStateException ise) {
  // Is the component CMI; is the service started and enabled?
}
myEjb.myBusinessMethod();
```

Within CMI application components, the Internationalization and InvocationInternationalization interfaces are semantically equivalent. You can use either of these interfaces to obtain the context associated with the thread on which that component is running. For instance, both interfaces can be used to obtain the list of locales propagated to the servlet doPost service method.

Internationalization context API: Programming reference

Application components programmatically manage internationalization context through the UserInternationalization, Internationalization, and InvocationInternationalization interfaces in the com.ibm.websphere.i18n.context package.

The following code example introduces the internationalization context API:

```
public interface UserInternationalization {
  public Internationalization getCallerInternationalization();
  public InvocationInternationalization
  getInvocationInternationalization();
}

public interface Internationalization {
  public java.util.Locale[] getLocales();
  public java.util.Locale getLocale();
  public java.util.TimeZone getTimeZone();
}

public interface InvocationInternationalization
    extends Internationalization {
  public void setLocales(java.util.Locale[] locales);
  public void setLocale(java.util.Locale jmLocale);
  public void setTimeZone(java.util.TimeZonetimeZone);
  public void setTimeZone(String timeZoneId);
}
```

UserInternationalization interface

The UserInternationalization interface provides factory methods for obtaining references to the CallerInternationalization and InvocationInternationalization context objects. Use these references to access elements of the caller and invocation contexts correlated to the current thread.

Methods of the UserInternationalization interface:

Internationalization getCallerInternationalization()
: Returns a reference implementing the Internationalization interface that supports access to elements of the caller internationalization context correlated to the current thread. If the service is disabled, this method issues an IllegalStateException exception.

InvocationInternationalization getInvocationInternationalization()
: Returns a reference implementing the InvocationInternationalization interface. If the service is disabled, this method issues an IllegalStateException exception.

Internationalization interface

The Internationalization interface declares methods that provide read-only access to internationalization context. Given a caller or invocation internationalization context object created with the UserInternationalization interface, bind the object to the Internationalization interface to get elements of that context type. Observe that caller internationalization context can be accessed only through this interface.

Methods of the Internationalization interface:

Locale[] getLocales()
: Returns the chain of locales within the internationalization context (object) that is bound to the interface, provided the chain is not null; otherwise this method returns a chain of length(1) containing the default locale of the Java virtual machine (JVM).

Locale getLocale()
: Returns the first in the chain of locales within the internationalization context (object) that is bound to the interface, provided the chain is not null; otherwise this method returns the default locale of the JVM.

TimeZone getTimeZone()

Returns the caller time zone (that is, the SimpleTimeZone instance) that is associated with the current thread, provided the time zone is non-null; otherwise this method returns the process time zone.

InvocationInternationalization interface

The InvocationInternationalization interface declares methods that provide read and write access to InvocationInternationalization context. Given an invocation internationalization context object created with the UserInternationalization interface, bind the object to the InvocationInternationalization interface to get and set elements of the invocation context.

According to the container-managed internationalization (CMI) policy, all set methods, setXxx(), issue an IllegalStateException exception when called within a CMI servlet or enterprise bean.

Methods of the InvocationInternationalization interface:

void setLocales(java.util.Locale[] locales)

Sets the chain of locales to the supplied chain, *locales*, within the invocation internationalization context. The supplied chain can be null or have length(>= 0). When the supplied chain is null or has length(0), the service sets the chain of invocation locales to an array of length(1) containing the default locale of the JVM. Null entries can exist within the supplied locale list, for which the service substitutes the default locale of the JVM on remote invocations.

void setLocale(java.util.Locale locale)

Sets the chain of locales within the invocation internationalization context to an array of length(1) containing the supplied locale, *locale*. The supplied locale can be null, in which case the service instead sets the chain to an array of length(1) containing the default locale of the JVM.

void setTimeZone(java.util.TimeZone timeZone)

Sets the time zone within the invocation internationalization context to the supplied time zone, *time zone*. If the supplied time zone is not an exact instance of java.util.SimpleTimeZone or is null, the service sets the invocation time zone to the default time zone of the JVM instead.

void setTimeZone(String timeZoneId)

Sets the time zone within the invocation internationalization context to the java.util.SimpleTimeZone having the supplied ID, *timeZoneId*. If the supplied time zone ID is null or invalid (that is, the ID is not displayed in the list of IDs returned by the java.util.TimeZone.getAvailableIds method) the service sets the invocation time zone to the simple time zone having an ID of GMT, an offset of 00:00, and otherwise invalid fields.

Internationalization context:

An *internationalization context* is a distributable collection of internationalization information containing an ordered list, or chain, of locales and a single time zone, where the locales and time zone are instances of the java.util.Locale and java.util.TimeZone Java SDK types, respectively. A locale chain is ordered according to the user's preference.

The internationalization service manages and makes available two varieties of internationalization context: the *caller context*, which represents the caller's localization environment, and the *invocation context*, which represents the localization environment under which a business method runs. Server application components use elements of the caller and invocation internationalization contexts to appropriately tailor locale-sensitive and time zone-sensitive computations.

The internationalization service does not support time zone types other than the java.util.SimpleTimeZone type that is found in the Java SDK. Unsupported time zone types silently map to the default time zone of the JVM when supplied to internationalization context API methods. For a complete description of the java.util.Locale, java.util.TimeZone and java.util.SimpleTimeZone types, refer the Java SDK API documentation.

Caller context

Caller internationalization context contains the locale chain and time zone received on incoming EJB business method and servlet service method invocations; it is the internationalization context propagated from the calling process. Use caller context elements within server application components to localize computations to the calling component. Caller context is read-only and can be accessed by all application components by using the Internationalization interface of the internationalization context API.

Caller context is computed in the following manner: On an EJB business method or servlet service method invocation, the internationalization service extracts the internationalization context from the incoming request and scopes this context to the method as the caller context. For any missing or null context element, the service inserts the corresponding default element of the JVM (for example, java.util.Locale.getDefault() or java.util.TimeZone.getDefault().) The service performs a similar insertion whenever missing or null Caller context elements are encountered on invocations of stateless session beans that are enabled for Web services.

Formally, caller context is the invocation context of the calling business method or application component.

Invocation context

Invocation internationalization context contains the locale chain and time zone under which EJB business methods and servlet service methods run. It is managed by either the hosting container or the application component, depending on the applicable internationalization policy. On outgoing business method requests, it is the context that propagates to the target process. Use invocation context elements to localize computations under the specified settings of the current application component.

Invocation context is computed in the following manner: On an incoming business method or servlet service method invocation, the internationalization service queries the associated context management policy. If the policy is container-managed internationalization (CMI), the container scopes the context designated by the policy to the invocation; otherwise the policy is application-managed internationalization (AMI), and the container scopes an empty context to the invocation that can be altered by the method implementation.

Application components can access invocation context elements through both the Internationalization and InvocationInternationalization interfaces of the internationalization context API. Invocation context elements can be set (overwritten) under the application-managed internationalization policy only.

On an outgoing business method request, the service obtains the currently scoped invocation context and attaches it to the request. This outgoing exported context becomes the caller context of the target invocation. When supplying invocation context elements, either for export on outgoing requests or through the API, the internationalization service always provides the most recent element set using the API; the service also supplies the corresponding default element of the JVM for any null invocation context element.

Because the internationalization context that is propagated over Web services (SOAP) requests contains a time zone ID rather than the entire state of a java.lang.SimpleTimeZone object, time zone information might be lost when a Web service-enabled client program or session bean becomes involved in remote business computation.

Internationalization context: Propagation and scope:

The scope of internationalization context is implicit. Every Enterprise JavaBeans (EJB) client application, servlet service method, and EJB business method call has two internationalization contexts under which it runs.

For each application component call, the container enters the caller context and the call context, as indicated by the pertinent internationalization policy, into scope before the container delegates to the actual implementation. When the implementation returns, the service removes these contexts from scope. The internationalization service supplies no programmatic mechanism for components to explicitly manage the scope of internationalization context.

The service scopes internationalization context differently with respect to application component type:
- "EJB client programs (contained)"
- "Servlets"
- "Enterprise beans"
- "Web service client programs (contained)" on page 563
- "Stateless session beans that are enabled for Web services" on page 563

Internationalization context observes by-value semantics over remote method requests. Changes to internationalization context elements that are scoped to a call do not affect the corresponding elements of the internationalization context that is scoped to the remote calling process. Also, modifications to context elements obtained using the internationalization context API do not affect the corresponding elements that are scoped to the invocation.

EJB client programs (contained)

Before it calls the main method of a client program, the Java EE client container introduces into scope invocation and caller internationalization some contexts that contain null elements. These contexts remain in scope throughout the life of the program. EJB client programs are the base in a chain of remote business method invocations and, technically, do not have a logical caller context. Accessing a caller context element yields the corresponding default element of the client JVM. On outgoing EJB business method requests, the internationalization service propagates the invocation context to the target process. Any unset (null) invocation context elements are replaced with the default of the JVM when exported by the internationalization context API or by outgoing requests.

Tip:

> To propagate values other than the JVM defaults to remote business methods, EJB client programs, as well as AMI servlets or enterprise beans, must set (override) elements of the invocation context. To learn how to set invocation context elements, see "Accessing invocation locales and time zones" on page 557.

Servlets

On every servlet service method (doGet or doPost) invocation, the Java EE Web container introduces caller and invocation internationalization contexts into scope before delegating to the service method implementation. The caller context contains the accept-languages propagated in the HTTP servlet request, typically from a Web browser. The invocation context contains whichever context is indicated by the container internationalization attribute of the internationalization policy that is associated with the servlet. Any unset (null) invocation context elements are replaced with the default of the server JVM when exported by the internationalization context API or by outgoing requests. The caller and invocation contexts remain effective until immediately after the implementation returns, at which time the container removes them from scope.

Enterprise beans

On every EJB business method invocation, the Java EE EJB container introduces caller and invocation internationalization contexts into scope before delegating to the business method implementation. The caller context contains the internationalization context elements imported from the incoming IIOP request; if the incoming request lacks a particular internationalization context element, the container scopes a null

element. The invocation context contains whichever context is indicated by the container internationalization attribute of the internationalization policy that is associated with the business method.

On outgoing EJB business method requests, the service propagates the invocation context to the target process. Any unset (null) invocation context elements are replaced with the default of the server JVM when exported by the internationalization context API or by outgoing requests. The caller and invocation contexts remain effective until immediately after the implementation returns, when the container removes them from scope.

Consider a simple EJB application with a Java client that calls the remote myBeanMethod bean method. On the client side, the application can use the Internationalization Service API to set invocation context elements. When the client calls myBeanMethod(), the service exports the client invocation context to the outgoing request. On the server side, the service detaches the imported context from the incoming request and scopes it to the method as its caller context; the service also scopes the invocation context to the method as indicated by the associated internationalization context management policy. The EJB container then calls the myBeanMethod method, which can use the internationalization context API to access elements of either the caller or invocation contexts. When the myBeanMethod method returns, the EJB container removes these contexts from scope.

Web service client programs (contained)

Before it calls the main method of a Web service client program, the client container introduces into scope both invocation and caller internationalization contexts that contain null elements. These contexts remain in scope throughout the duration of the program. Web service client programs are the base in a chain of remote business method invocations and, technically, do not have a logical caller context. Accessing a Caller context element yields the corresponding default element of the client virtual machine.

On outgoing Web service requests, the internationalization service transparently creates a SOAP header block that contains the invocation context that is associated with the current thread; the SOAP representation of invocation context is propagated through the request to the target process. Any unset (that is, null) invocation context elements are replaced with the default element of the JVM when exported by the internationalization context API or by outgoing requests. Also, because the header contains only a time zone ID, the additional state of the time zone object (java.lang.SimpleTimeZone) of the invocation context might be lost, because it does not get propagated through the request.

Tip:

> To propagate values other than the JVM defaults to remote business methods, Web service client programs, as well as AMI servlets or enterprise beans, must set (override) elements of the invocation context. For more information, see “Accessing invocation locales and time zones” on page 557.

Stateless session beans that are enabled for Web services

On every method invocation of a Web service-enabled bean, the EJB container introduces caller and invocation internationalization contexts into scope before delegating control to the business method implementation. The caller context contains the internationalization context elements that are imported from the SOAP header block of the incoming request. If the incoming request lacks a particular internationalization context element, the container introduces a null element into scope. The invocation context contains whichever context is indicated by the container internationalization attribute of the internationalization policy that is associated with the business method.

On outgoing EJB business method requests, the service propagates the invocation context to the target process. Any unset (that is, null) invocation context elements are replaced with the default element of the server JVM when exported by the internationalization context API or by outgoing requests. The caller and invocation contexts remain effective until immediately after control returns from the business method implementation, at which time the container removes them from scope.

On outgoing Web service requests, the internationalization service transparently creates a SOAP header block that contains the invocation context associated with the current thread. The SOAP representation of the invocation context is propagated through the request to the target process. Any unset (that is, null) invocation context elements are replaced with the default element of the JVM when exported by the internationalization context API or by outgoing requests.

Thread association considerations

The Web and EJB containers scope internationalization contexts to a method by associating the method with the thread that run the method implementation. Similarly, methods of the internationalization context API either associate context with, or obtain context associated with, the thread on which these methods run.

In cases where new threads are spawned within an application component (for instance, a user-generated thread inside the service method of a servlet, or a system-generated event handling thread in an AWT client) the internationalization contexts associated with the parent thread does not automatically transfer to the newly-spawned thread. In such instances, the service exports the default locale and time zone of the JVM on any remote business method request and on any API calls that run on the new thread.

If the default context is inappropriate, the desired invocation context elements must be explicitly associated to the new thread by using the setXxx methods of the InvocationInternationalization interface. Currently, internationalization context management policies enable invocation context to be set within EJB client programs, as well as within servlets, session beans, and message-driven beans that use application-managed internationalization.

Internationalization context: Management policies:

Internationalization policies prescribe how Java EE application components or their hosting containers manage internationalization context on component invocations. Two internationalization context management policies apply to all component types: Application-managed internationalization (AMI) and Container-managed internationalization (CMI).

These policies are represented in two parts:
- Internationalization type
- Container internationalization attribute

The service defines a default, or implicit, internationalization policy for every application component type. At development time, assemblers can override the default policy for server component types by explicitly configuring their internationalization type, and optional container internationalization attributes. Policies configured during assembly are preserved in the deployment descriptor for the application.

All components have an internationalization type that indicates whether it is AMI or CMI; that is, whether a component is to deploy under the application-managed or the container-managed internationalization policy. Application assemblers can set the internationalization type for servlets, session beans, and message-driven beans. Entity beans are implicitly CMI and EJB clients are implicitly AMI; neither can be configured otherwise.

For CMI servlets and enterprise beans, optional container internationalization attributes can be specified to indicate which invocation internationalization context the container is to scope to service or business methods. A CMI service or business method invocation can run under the context of the caller's process, under the default context of the server JVM, or under a custom context specified in the attribute. Assemblers can specify one container internationalization attribute per disjoint set of CMI servlets within a Web module, or one Attribute per disjoint set of business methods of CMI beans within an EJB module. A container internationalization attribute can be associated with more than one method, but a method cannot be associated with more than one attribute.

When an application server launches an application, the internationalization service collects policy information from the deployment descriptor, then uses this information to construct and associate an internationalization policy to every component invocation. A policy is denoted as:

```
[<Internationalization Type>,<Container Internationalization Attribute>]
```

Several cases exist in which the deployment descriptor seems to lack policy information, for example: EJB client applications have no configurable internationalization policy settings; AMI components do not have container internationalization attributes; and you are not required to specify container internationalization attributes for CMI components. When the service cannot obtain the explicit internationalization type and container attribute settings from a well-formed deployment descriptor, it implicitly inserts the appropriate setting into the policy.

The service observes the following conventions when applying policies to invocations:

- Servlets (service) and EJB business methods lacking all internationalization policy information in the deployment descriptor implicitly run under policy `[CMI,RunAsCaller]`.
- CMI servlets and business methods lacking a container internationalization attribute in the deployment descriptor implicitly run under policy `[CMI,RunAsCaller]`.
- AMI servlets and business methods always lack container internationalization attributes in the deployment descriptor, but implicitly run under the logical policy `[AMI,RunAsServer]`.
- EJB clients always lack internationalization policy information in the deployment descriptor. By definition, EJB clients are implicitly AMI types and run under the invocation context of the JVM; they run under the logical policy `[AMI,RunAsServer]`.

For conditions other than these cited examples, such as a malformed deployment descriptor, refer to Internationalization service errors.

Internationalization policies for EJB clients and HTTP clients cannot be configured; HTTP clients do, however, run under the language priority settings of the hosting Web browser. These settings are configurable under the options dialog of most Web browsers. Refer to your Web browser documentation for details.

Internationalization type:

Every server application component has an *internationalization type* setting that indicates whether the invocation internationalization context is managed by the component or by the hosting Java EE container.

Server application components can be deployed to use one of two types of internationalization context management:

- Application-managed internationalization (AMI)
- Container-managed internationalization (CMI)

A server component can be deployed as AMI or CMI, but not both; CMI is the default. The setting applies to the entire component on every invocation. Entity beans use CMI only. Enterprise JavaBeans (EJB) client applications do not have an internationalization type setting; they implicitly use AMI.

Application-managed internationalization

Under the AMI deployment policy, component developers assume complete control over the invocation internationalization context. AMI components can use the internationalization context API to programmatically set invocation context elements.

AMI components are expected to manage invocation context. Invocations of AMI components implicitly run under the default locale and time zone of the hosting JVM. Invocation context elements not set using the API default to the corresponding elements of the JVM when accessed through the API or when exported on business methods. To export context elements other than the JVM defaults, AMI servlets, AMI enterprise beans, and EJB client applications must set (overwrite) invocation elements using the

internationalization context API. Moreover, the container logically suspends the caller context that is imported on the AMI servlet lifecycle method and AMI EJB business method invocations. To continue propagating the context of the calling process, AMI servlets and enterprise beans must use the API to transfer caller context elements to the invocation context.

Specify AMI for server components that have internationalization context management requirements that are not supported by container-managed internationalization (CMI).

Container-managed internationalization

CMI is the preferred internationalization context management policy for server application components; it is also the default policy. Under CMI, the internationalization service collaborates with the Web and EJB containers to set the invocation internationalization context for servlets and enterprise beans. The service sets invocation context according to the container internationalization attribute of the policy that is associated with a servlet (service method) or an EJB business method.

A CMI policy has a container internationalization attribute that indicates which internationalization context the container is to scope to an invocation. For details, see Container internationalization attributes. By default, invocations of CMI components run under the caller's internationalization context; or rather, they adhere to the implicit policy `[CMI,RunasCaller]` whenever the servlet or business is not associated with an attribute in the deployment descriptor. For complete details, see Internationalization context: Management policies.

Methods within CMI components can obtain elements of the invocation context using the internationalization context API, but cannot set them. Any attempt to set invocation context elements within CMI components results in a java.lang.IllegalStateException exception.

Specify container-managed internationalization for server application components that require standard internationalization context management. Then specify the container internationalization attributes for CMI servlets and for business methods of CMI enterprise beans that you do not want to run under the caller's internationalization context.

Container internationalization attributes:

The internationalization policy of every CMI servlet and EJB business method has a *container internationalization attribute* that specifies which internationalization context the container is to scope to its invocation.

The container internationalization attribute has three main fields:
- Run as
- Locales
- Time zone ID

As a convenience, you can create named container internationalization attributes and associate them to the following subsets:
- CMI servlets within a Web module
- Business methods of CMI enterprise beans within an Enterprise JavaBeans (EJB) module
- Business methods of Web service-enabled session beans. In the following descriptions, the term *supported enterprise bean* refers to both CMI enterprise beans and Web service-enabled session beans.

Run-as field

The **Run-as** field specifies one of three types of invocation context that a container can scope to a method. For servlet service and EJB business methods, the container constructs the invocation

internationalization context according to the **Run as** field setting and associates this context to the current thread before delegating to the method implementation.

By default, invocations of servlet service methods and EJB business methods implicitly run as caller (`RunAsCaller`) unless the **Run as** field of a policy attribute specifies otherwise. EJB client applications and AMI server components always run as server (`RunAsServer`).

You can specify the following invocation context types with the **Run as** field are:

Caller The container calls the method under the internationalization context of the calling process. For any missing context element, the container supplies the corresponding default context element of the Java virtual machine (JVM). Select run as caller when you want the invocation to run under the invocation context of the calling process.

Server
The container calls the method under the default locale and time zone of the JVM. Select run as server when you want the invocation to run under the invocation context of the JVM.

Specified
The container calls the method under the internationalization context specified in the attribute. Select run as specified when you want the invocation to run under the custom invocation context that is specified in the policy; then provide the custom context elements by completing the Locales and Time zone ID fields.

Remember: Java Message Service (JMS) messages do not contain internationalization context. Although container-managed message-driven beans can be configured to run as caller, the container associates the default elements of the server process when calling the onMessage method of any message-driven bean that is configured as `[CMI, RunAsCaller]`. You can also configure the **Run as** field for Web service business methods.

Locales field

The **Locales** field specifies an ordered list of locales that the container scopes to an invocation. A locale represents a specific geographical, cultural, or political region and contains three fields:

- **Language code**. Ideally, language code is one of the lower-case, two-character codes that are defined by the ISO 639 standard; however, language code is not restricted to ISO codes and is not a required field. A valid locale must specify a language code if it does not specify a country code.
- **Country code**. Ideally, country code is one of the upper-case, two-character codes that are defined by the ISO 3166 standard; however, country code is not restricted to ISO codes and is not a required field. A valid locale must specify a country code if it does not specify a language code.
- **Variant**. Variant is a vendor-specific code. Variant is not a required field and serves only to supplement the language and country code fields according to application- or platform-specific requirements.

A valid locale must specify at least a language code or a country code; the variant is always optional. The first locale of the list is returned when accessing invocation context using the getLocale method of the internationalization context API.

Time zone ID field

The **Time zone ID** field specifies an abbreviated identifier for a time zone that the container scopes to an invocation. You can also configure the **Time zone ID** field for Web service business methods.

A time zone represents a temporal offset and computes daylight savings information. A valid ID indicates any time zone supported by the java.util.TimeZone type. Specifically, a valid ID is any of the IDs that appear in the list of time zone IDs returned by method java.util.TimeZone.getAvailableIds(), or a custom ID having the form `GMT[+|-]hh[[:]mm]`; for example, `America/Los_Angeles`, `GMT-08:00` are valid time zone IDs.

Administering the internationalization service

To use internationalization context in an Enterprise JavaBeans (EJB) application, the internationalization service must be enabled in the runtime environments for all server-side components (servlets and enterprise beans, including session beans enabled for Web service usage) as well as all client-side components (EJB client applications and Web service clients).

About this task

If you do not require the internationalization service, do not enable it. Leaving the service disabled prevents any possible performance degradation incurred by the implicit distribution of internationalization resources.

The internationalization service cannot be enabled for HTTP clients, because support for internationalization in that case is provided by the browser, not by the application server.

Procedure

- Enable or disable the internationalization service for servlets and enterprise beans. By default, the service is disabled for server-side components within the application server. You enable the service by using either the administrative console or the wsadmin tool.
- Enable or disable the internationalization service for EJB clients. By default, the service is disabled within the client container. You enable the service by using the launchClient tool.

Enabling the internationalization service for servlets and enterprise beans

Perform this task to enable the internationalization service in the application server runtime environment.

About this task

Any servlet or enterprise bean can use internationalization context if the internationalization service is enabled within the hosting application server instance.

Procedure

1. Start the administrative console.
2. Click **Servers** > **Application servers** > ***server_name*** > **Container services** > **Internationalization service**.
3. Enable the internationalization service.
   a. If not already selected, select the **Enable service at server startup** check box.
   b. Click **OK**.

Results

When you select the **Enable service at server startup** setting, the application server automatically initializes and starts the internationalization service whenever the server starts. If you change this setting, be sure to restart the application server for the new setting to take effect.

To disable the service, clear the **Enable service at server startup** check box. In this case, the internationalization service is initialized but not started when the application server starts.

Example

Alternatively, the internationalization service can be enabled from the command line by using the wsadmin tool. Start the wsadmin tool and enter the following commands:

```
set x [$AdminConfig list I18NService]
$AdminConfig modify $x { { enable true } }
$AdminConfig save
exit
```

What to do next

If you enable or disable the internationalization service, be sure to stop and then restart the application server for the new setting to take effect.

Enabling the internationalization service for EJB clients

By default, the internationalization service is disabled for use within Enterprise JavaBeans (EJB) and Web-service enabled client applications. You must enable the service for client applications as well as for all server instances in the runtime environment.

Procedure

Enable the service.
When calling the launchClient tool, include the argument `-CCDI18NService.enable=true` or `-CCDI18NService.enable=yes`.

Internationalization service settings

Use this page to enable or disable the internationalization service. The internationalization service manages the implicit propagation and scoping of locale and time zone information, called *internationalization context*, within application components. When the service is enabled, application components can use the internationalization context API to programmatically manage locale and time zone information. In turn, components can use that locale and time zone information with the Java Platform, Standard Edition (JSE) Internationalization API to perform localizations. If internationalization support is not required on the server, disabling the service can improve performance.

To view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name*. Then, under Container Settings, click **Container Services > Internationalization Service**.

Enable service at server startup:

Specifies whether the server attempts to start the internationalization service.

Information	Value
Default	Cleared
Range	Valid values are Selected or Cleared

More information about valid values follows:

Selected
When the application server starts, it attempts to start the internationalization service automatically.

Cleared
The server does not try to start the internationalization service.

To enable the internationalization service for applications on this server, the system administrator must select this property and then restart the server.

Internationalization service errors

Certain conditions might cause the internationalization service not to start, to issue java.lang.IllegalStateException exceptions while an application is running, or to exercise default behaviors.

The java.lang.IllegalStateException exception indicates one of the following things:

- An application component attempted an operation that is not supported by the internationalization programming model.

  The IllegalStateException exception is issued whenever a server application component whose internationalization type is set to container-managed internationalization (CMI) attempts to set invocation context. This behavior is a violation of the CMI policy, under which servlets and enterprise beans cannot modify their invocation internationalization context.
- An anomaly occurred that disabled the service.

  For instance, if the internationalization service is not properly initialized, the Java Naming and Directory Interface (JNDI) lookup on the UserInternationalization URL attribute issues a javax.naming.NameNotFoundException exception that contains an IllegalStateException instance.

The following conditions can occur while your internationalized application is running. These conditions might cause the internationalization service not to start, to issue IllegalStateException exceptions, or to exercise default behaviors:
- "The service is disabled "
- "The service is not started"
- "Invalid context element" on page 571
- "Missing context element" on page 572
- "Invalid policy" on page 572
- "Missing policy" on page 572

If you encounter unexpected or exceptional behavior, the problem is likely related to one of these conditions. You need to examine the trace log to investigate these conditions, which requires that you configure the diagnostic trace service to generate messages about internationalization service function.

The trace strings for the internationalization service follow; use both:

```
com.ibm.ws.i18n.context.*=all=enabled:com.ibm.websphere.i18n.context.*=all=enabled
```

The service is disabled

The internationalization service is not initialized when the startup setting is cleared. The service generates a message that indicates whether it is enabled or disabled. Applications cannot access the internationalization API when the service is disabled. If an application attempts a JNDI lookup to obtain the UserInternationalization reference, the lookup fails with a NamingException exception, indicating that the reference cannot be found. In addition, the service does not scope (propagate) internationalization context on incoming (outgoing) business method calls.

The service is not started

The internationalization service is operational whenever it is in the STARTED state. For example, if an application attempts to access internationalization context and the service is not started, the API issues an IllegalStateException exception. In addition, the service does not provide runtime support for servlets and enterprise beans.

As an application server progresses through its life cycle, it initializes, starts, stops, and terminates (destroys) the internationalization service. If an anomaly occurs during initialization, the service does not start. After the service is started, its state can change to BLOCKED in the event that a serious error occurs. The service generates a message for every state change.

If a trace message indicates that the service is not STARTED, examine previous messages to determine the problem. For instance, the internationalization service does not start if the activity service is unavailable and a message is displayed to that effect during initialization of the internationalization service.

During startup, the following messages indicate potential configuration or runtime problems:

No ORB support
The service cannot obtain an instance of the object request broker (ORB). This condition is a fatal error. Examine the `SystemErr.log` and `SystemOut.log` files for information.

Note: This topic references one or more of the application server log files. As a recommended alternative, you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files on distributed and IBM i systems. You can also use HPEL in conjunction with your native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.

No TCM support
The service cannot obtain an instance of its thread context manager (TCM). This condition is a fatal error. Examine the `SystemErr.log` and `SystemOut.log` files for information.

No IIOP (activity service) support
The service cannot register with the activity service. This condition is a fatal error. The internationalization service cannot propagate or receive context on Internet Inter-ORB Protocol (IIOP) requests without activity service support. Examine the `SystemErr.log` and `SystemOut.log` files for information.

No AsynchBeans support
The service cannot register into the asynchronous beans environment. This warning indicates that the asynchronous beans environment cannot support internationalization context.

No EJB container support
The service cannot register with the Enterprise JavaBeans (EJB) container. This warning indicates that the internationalization service cannot support enterprise beans. Without EJB container support, internationalization contexts do not scope properly to EJB business methods. Review the trace log for any EJB container-related error conditions.

No Web container support
The service cannot register with the Web container. This warning indicates that the internationalization service cannot support servlets and JavaServer Page (JSP) files. Without Web container support, internationalization contexts do not scope properly to servlet service methods. Review the trace log for any Web container-related error conditions.

No Metadata support
The service cannot register with the metadata service. This warning indicates that the internationalization service cannot process the internationalization policies within application deployment descriptors. Without metadata support, the service associates the default internationalization context management policy, [CMI, RunAsCaller], to every servlet lifecycle method and enterprise bean business method invocation. Review the trace log for any metadata service-related error conditions.

No JNDI (Naming service) support
The service cannot bind the UserInternationalization object into the namespace. This condition is a fatal error. Application components are unable to access internationalization context API references, and are therefore unable to access internationalization context elements. Review the trace log for any Naming (JNDI) service-related error conditions.

No API support
The service cannot obtain an instance of an internationalization context API object. This condition is a fatal error. Application components are unable to access internationalization context API references, and are therefore unable to access internationalization context elements.

Invalid context element

The service detected an invalid internationalization context element. For example, the internationalization service does not support TimeZone instances of a type other than java.util.SimpleTimeZone. If the service encounters an unusable element, it logs a message and substitutes the corresponding default element of the JVM.

Missing context element

The service detected a missing internationalization context element. Incoming requests (for example, from application servers that do not support the internationalization service) lack internationalization context. When the service attempts to access a caller internationalization context element (which does not exist in this case), the service logs a message and substitutes the corresponding default element of the Java virtual machine (JVM).

Whenever possible, enable the internationalization service within all clients and hosting application servers that comprise an internationalized enterprise application. Read more information about Administering the internationalization service in the *Administering applications and their environment* PDF book.

Invalid policy

The internationalization service detected a malformed internationalization policy in the application deployment descriptor. The service replaces the malformed attribute with the appropriate default. For instance, if the internationalization type for an entity bean is set to `Application` during the run of a servlet or EJB business method call, the service logs the inconsistency and enforces the `Container` setting instead.

Also, AMI application components do have an implicit container internationalization attribute. By default they run as server. The service silently enforces the implicit policy, `[AMI, RunAsServer]`, and logs messages to this effect.

Invalid container internationalization attributes are likely to occur when specifying the Locales and Time zone ID fields. When encountering invalid locales and time zone IDs within attributes, the service replaces each value with the corresponding default element of the JVM. Be sure to follow the guidelines provided in the *Developing and deploying applications* PDF book.

Missing policy

The service detected a missing internationalization policy. The service replaces the missing policy with the appropriate default. For instance, if the internationalization type is missing for a servlet or enterprise bean, the service sets the attribute to Container.

Container internationalization attributes are not mandatory for CMI application components. In the event that a CMI servlet or EJB business method lacks a container internationalization attribute, the service silently enforces the implicit policy `[CMI, RunAsCaller]`.

When an application lacks internationalization policies in its deployment descriptor, or metadata support is unavailable, the service logs a message and applies the policy `[CMI, RunAsCaller]` on every servlet service method and EJB business method invocation.

Read the information in the *Developing and deploying applications* PDF book:
- Assembling internationalized applications
- Container internationalization attributes
- Internationalization type

Chapter 12. Developing Mail, URLs, and other Java EE resources

This page provides a starting point for finding information about resources that are used by applications that are deployed on a Java Enterprise Edition (Java EE)-compliant application server. They include:

- JavaMail support for applications to send Internet mail
- URLs, for describing logical locations
- Resource environment entries, for mapping logical names to physical names
- Java DataBase Connectivity (JDBC) resources and other technology for data access (discussed elsewhere)
- Java Message Service (JMS) resources and other messaging system support (discussed elsewhere)

Developing applications that use the JavaMail API

JavaMail API

The JavaMail APIs provide a framework that is platform and protocol independent for building mail client applications that are based on Java. The JavaMail APIs are generic for sending and reading mail. They require service providers, known in the application server as protocol providers, to interact with mail servers that run on pertaining protocols. For example, Simple Mail Transfer Protocol (SMTP) is a popular transport protocol for sending mail. Mail applications can connect to an SMTP server and send mail through it by using this SMTP protocol provider.

The application server supports the JavaMail API, Version 1.4. In the application server, the JavaMail API is supported in all web application components, namely:

- servlets
- JavaServer Pages (JSP) files
- enterprise beans
- application clients

In addition to service providers, the JavaMail API requires the JavaBeans Activation Framework (JAF) to handle mail content that is not plain text, including Multipurpose Internet Mail Extensions (MIME), URL pages, and file attachments.

The JavaMail APIs, the JAF, the service providers, and the protocols are shipped as part of the application server. The API and related specifications are repackaged from materials that are licensed.

Note: If you are using Java 5 to run your code, you will need the JAF 1.1 package. For Java 6 and later, the JAF package is part of the run time environment.

Debugging mail sessions

When you debug a mail application, you can use the mail debugging feature. The mail component generates debugging information, on a per session basis, that can be used for problem determination or tuning.

About this task

Enabling the debug mode triggers the mail component of the application server to print the following data to the standard output stream:

- interactions with the mail servers
- properties of the mail session

This output stream is redirected to the SystemOut.log file for the specific application server.

Note: This topic references one or more of the application server log files. As a recommended alternative, you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using SystemOut.log , SystemErr.log, trace.log, and activity.log files on distributed and IBM i systems. You can also use HPEL in conjunction with your native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.

Procedure

1. Open the administrative console.
2. Click **Resources** > **Mail** > **Mail sessions** > ***mail session***.
3. Click **Enable debug mode**. Debugging is enabled for that session only.
4. Click **Apply** or **OK**.

Example

The following example shows sample mail debugging output:

```
ResourceMgrIm I   WSVR0049I: Binding Test as mail/test
SystemOut     O *** In SessionReferenceable.getReference:
SystemOut     O  added StringRefAddr: type=ws.transport.password, content=****
SystemOut     O  added StringRefAddr: type=ws.isolated.class.loader, content=false
SystemOut     O  added StringRefAddr: type=mail.transport.protocol, content=smtp
SystemOut     O  added StringRefAddr: type=mail.imaps.class, content=com.sun.mail.imap.IMAPSSLStore
SystemOut     O  added StringRefAddr: type=mail.smtp.host, content=smtp.coldmail.com
SystemOut     O  added StringRefAddr: type=mail.debug, content=true
SystemOut     O  added StringRefAddr: type=mail.pop3s.class, content=com.sun.mail.pop3.POP3SSLStore
SystemOut     O  added StringRefAddr: type=mail.from, content=smith@coldmail.com
SystemOut     O  added StringRefAddr: type=mail.smtp.class, content=com.sun.mail.smtp.SMTPTransport
SystemOut     O  added StringRefAddr: type=mail.smtps.class, content=com.sun.mail.smtp.SMTPSSLTransport
SystemOut     O  added StringRefAddr: type=mail.imap.class, content=com.sun.mail.imap.IMAPStore
SystemOut     O  added StringRefAddr: type=mail.smtp.user, content=smith
SystemOut     O  added StringRefAddr: type=mail.pop3.class, content=com.sun.mail.pop3.POP3Store
SystemOut     O  added StringRefAddr: type=mail.mime.address.strict, content=true


SystemOut     O DEBUG: JavaMail version 1.4ea
SystemOut     O DEBUG: java.io.FileNotFoundException:
C:\Program Files\IBM\WebSphere\AppServer\java\jre\lib\javamail.providers
(The system cannot find the file specified.)
SystemOut     O DEBUG: !anyLoaded
SystemOut     O DEBUG: not loading resource: /META-INF/javamail.providers
SystemOut     O DEBUG: successfully loaded resource: /META-INF/javamail.default.providers
SystemOut     O DEBUG: Tables of loaded providers
SystemOut     O DEBUG: Providers Listed By Class Name:
     {com.sun.mail.smtp.SMTPSSLTransport=javax.mail.Provider
[TRANSPORT,smtps,com.sun.mail.smtp.SMTPSSLTransport,Sun Microsystems, Inc],
com.sun.mail.smtp.SMTPTransport=javax.mail.Provider
[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun Microsystems, Inc],
com.sun.mail.imap.IMAPSSLStore=javax.mail.Provider
[STORE,imaps,com.sun.mail.imap.IMAPSSLStore,Sun Microsystems, Inc],
com.sun.mail.pop3.POP3SSLStore=javax.mail.Provider
[STORE,pop3s,com.sun.mail.pop3.POP3SSLStore,Sun Microsystems, Inc],
com.sun.mail.imap.IMAPStore=javax.mail.Provider
[STORE,imap,com.sun.mail.imap.IMAPStore,Sun Microsystems, Inc],
com.sun.mail.pop3.POP3Store=javax.mail.Provider
[STORE,pop3,com.sun.mail.pop3.POP3Store,Sun Microsystems, Inc]}
SystemOut     O DEBUG: Providers Listed By Protocol:
{imaps=javax.mail.Provider[STORE,imaps,com.sun.mail.imap.IMAPSSLStore,Sun Microsystems,Inc],
imap=javax.mail.Provider[STORE,imap,com.sun.mail.imap.IMAPStore,Sun Microsystems, Inc],
smtps=javax.mail.Provider[TRANSPORT,smtps,com.sun.mail.smtp.SMTPSSLTransport,Sun Microsystems,Inc],
pop3=javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Sun Microsystems, Inc],
pop3s=javax.mail.Provider[STORE,pop3s,com.sun.mail.pop3.POP3SSLStore,Sun Microsystems, Inc],
smtp=javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Sun Microsystems, Inc]}
SystemOut     O DEBUG: successfully loaded resource: /META-INF/javamail.default.address.map
SystemOut     O DEBUG: !anyLoaded
SystemOut     O DEBUG: not loading resource: /META-INF/javamail.address.map
SystemOut     O DEBUG: java.io.FileNotFoundException:
     C:\Program Files\IBM\WebSphere\AppServer\java\jre\lib\javamail.address.map
     (The system cannot find the file specified.)
SystemOut     O *** In SessionFactory.setPasswordAuthentication,
   TRANSPORT PasswordAuthentication is based on:
SystemOut     O  url=smtp://smith@smtp.coldmail.com
SystemOut     O  user=smith
SystemOut     O  password=****
```

```
SystemOut     O *** In SessionFactory.getObjectInstance, session properties:
SystemOut     O  mail.transport.protocol=smtp
SystemOut     O  mail.imaps.class=com.sun.mail.imap.IMAPSSLStore
SystemOut     O  mail.smtp.host=smtp.coldmail.com
SystemOut     O  mail.debug=true

SystemOut     O  mail.pop3s.class=com.sun.mail.pop3.POP3SSLStore
SystemOut     O  mail.from=smith@coldmail.com
SystemOut     O  mail.smtp.class=com.sun.mail.smtp.SMTPTransport
SystemOut     O  mail.smtps.class=com.sun.mail.smtp.SMTPSSLTransport
SystemOut     O  mail.imap.class=com.sun.mail.imap.IMAPStore
SystemOut     O  mail.smtp.user=smith
SystemOut     O  mail.pop3.class=com.sun.mail.pop3.POP3Store
SystemOut     O  mail.mime.address.strict=true
SystemOut     O DEBUG: mail.smtp.class property exists and points to com.sun.mail.smtp.SMTPTransport
SystemOut     O DEBUG SMTP: useEhlo true, useAuth false
SystemOut     O DEBUG SMTP: trying to connect to host "smtp.coldmail.com", port 25, isSSL false


javax.mail.MessagingException: Unknown SMTP host: smtp.coldmail.com;
  nested exception is:
 java.net.UnknownHostException: smtp.coldmail.com
 at com.sun.mail.smtp.SMTPTransport.openServer(SMTPTransport.java:1280)
 at com.sun.mail.smtp.SMTPTransport.protocolConnect(SMTPTransport.java:370)
 at javax.mail.Service.connect(Service.java:275)
 at javax.mail.Service.connect(Service.java:156)
 at javax.mail.Service.connect(Service.java:105)
 at javax.mail.Transport.send0(Transport.java:168)
 at javax.mail.Transport.send(Transport.java:98)
 at com.ibm.ws.mail.ut.TestServlet.doTask(TestServlet.java:104)
 at com.ibm.ws.mail.ut.TestServlet.doGet(TestServlet.java:65)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:707)
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:820)
 at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:1397)
 at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:759)
 at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:429)
 at com.ibm.ws.webcontainer.servlet.ServletWrapperImpl.handleRequest(ServletWrapperImpl.java:175)
 at com.ibm.ws.webcontainer.webapp.WebApp.handleRequest(WebApp.java:3512)
 at com.ibm.ws.webcontainer.webapp.WebGroup.handleRequest(WebGroup.java:273)
 at com.ibm.ws.webcontainer.WebContainer.handleRequest(WebContainer.java:896)
 at com.ibm.ws.webcontainer.WSWebContainer.handleRequest(WSWebContainer.java:1530)
 at com.ibm.ws.webcontainer.channel.WCChannelLink.ready(WCChannelLink.java:161)
 at com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.handleDiscrimination(HttpInboundLink.java:455)
 at com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.handleNewInformation(HttpInboundLink.java:384)
 at com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.ready(HttpInboundLink.java:272)
 at com.ibm.ws.tcp.channel.impl.NewConnectionInitialReadCallback.sendToDiscriminators(NewConnectionInitialReadCallback.java:214)
 at com.ibm.ws.tcp.channel.impl.NewConnectionInitialReadCallback.complete(NewConnectionInitialReadCallback.java:113)
 at com.ibm.ws.tcp.channel.impl.AioReadCompletionListener.futureCompleted(AioReadCompletionListener.java:165)
 at com.ibm.io.async.AbstractAsyncFuture.invokeCallback(AbstractAsyncFuture.java:217)
 at com.ibm.io.async.AsyncChannelFuture.fireCompletionActions(AsyncChannelFuture.java:161)
 at com.ibm.io.async.AsyncFuture.completed(AsyncFuture.java:138)
 at com.ibm.io.async.ResultHandler.complete(ResultHandler.java:202)
 at com.ibm.io.async.ResultHandler.runEventProcessingLoop(ResultHandler.java:766)
 at com.ibm.io.async.ResultHandler$2.run(ResultHandler.java:896)
 at com.ibm.ws.util.ThreadPool$Worker.run(ThreadPool.java:1487)
Caused by: java.net.UnknownHostException: smtp.coldmail.com
 at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:196)
 at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:366)
 at java.net.Socket.connect(Socket.java:519)
 at java.net.Socket.connect(Socket.java:469)
 at com.sun.mail.util.SocketFetcher.createSocket(SocketFetcher.java:232)
 at com.sun.mail.util.SocketFetcher.getSocket(SocketFetcher.java:189)
 at com.sun.mail.smtp.SMTPTransport.openServer(SMTPTransport.java:1250)
 ... 32 more
```

This output illustrates a connection failure to a Simple Mail Transfer Protocol (SMTP) server because a fictitious name, smtp.coldmail.com, is specified as the server name.

The following list provides tips on reading the previous sample of debugger output:

- The lines headed by `DEBUG` are printed by the mail provider at run time, while the two lines headed by *** are printed by the application server at run time.
- In the second paragraph of code, the first few lines state that some configuration files are skipped. The mail component attempts to load a number of configuration files from different locations at run time. All those files are not required. If a required file cannot be accessed, however, the mail component creates an exception. In this sample, there is no exception and the third-line announces that default providers are loaded.
- The next few lines, headed by either `Providers Listed by Class Name` or `Providers Listed by Protocols`, show the protocol providers that are loaded. The six providers that are listed are the default

protocol providers that come under the built-in mail provider for the application server. If you install special service providers, and these providers are used in the current mail session, those providers are listed here with the default providers.

- The two lines headed by *** and the few lines below them are printed by the application server to show the configuration properties of the current mail session. Although these properties are listed by their internal name rather than the name you establish in the administrative console, you can easily recognize the relationships between them. For example, the mail.store.protocol property corresponds to the **Protocol** property in the **Incoming Mail Properties** section of the console panel for mail session configuration. Review the listed properties and values to verify that they correspond.
- The few lines above the exception stack show the mail activities when sending a message. First, the JavaMail API recognizes that the transport protocol is set to SMTP and that the com.sun.mail.smtp.SMTPTransport provider exists. Next, the output log displays the useEhlo and useAuth parameters, which are used by SMTP. Finally, the log shows the SMTP provider trying to connect to the smtp.coldmail.com mail server.
- The output log show the exception stack next. This data indicates that the specified mail server either does not exist or is not functioning.

Chapter 13. Developing Messaging resources

This page provides a starting point for finding information about the use of asynchronous messaging resources for enterprise applications with WebSphere Application Server.

WebSphere Application Server supports asynchronous messaging based on the Java Message Service (JMS) and the Java EE Connector Architecture (JCA) specifications, which provide a common way for Java programs (clients and Java EE applications) to create, send, receive, and read asynchronous requests, as messages.

JMS support enables applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). Some messaging providers also allow WebSphere Application Server applications to use JMS support to exchange messages asynchronously with non-JMS applications; for example, WebSphere Application Server applications often need to exchange messages with WebSphere MQ applications. Applications can explicitly poll for messages from JMS destinations, or they can use message-driven beans to automatically retrieve messages from JMS destinations without explicitly polling for messages.

WebSphere Application Server supports the following messaging providers:

- The WebSphere Application Server default messaging provider (which uses service integration as the provider).
- The WebSphere MQ messaging provider (which uses your WebSphere MQ system as the provider).
- Third-party messaging providers that implement either a JCA Version 1.5 resource adapter or the ASF component of the JMS Version 1.0.2 specification.

Programming to use asynchronous messaging

You can build enterprise applications that use Java Message Service (JMS) APIs directly to provide asynchronous messaging services. You can also use message-driven beans as asynchronous message consumers. If you are writing messaging programs that interoperate between WebSphere Application Server and WebSphere MQ, there are some environmental differences that you need to take into account.

About this task

Enterprise applications can use JMS APIs directly to explicitly poll for messages on a JMS destination, then retrieve messages for processing by business logic beans (enterprise beans).

Message-driven beans can also be used as asynchronous message consumers. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination.

Procedure

- “Programming to use JMS and messaging directly” on page 578.

  Your enterprise applications can use Java Message Service (JMS) programming interfaces directly to provide messaging services, and methods that implement business logic.

  If you choose not to use JNDI to obtain configuration information for your messaging provider, for example for connection factories or destinations, you can instead use an API provided by your messaging provider to specify that configuration information programmatically.
- “Programming for interoperation with WebSphere MQ” on page 593

  There are some differences between the WebSphere Application Server environment and the WebSphere MQ environment. If you are writing messaging programs that interoperate between these two environments, you should be aware of these differences and take them into account when designing, coding and deploying your programs.

- “Programming to use message-driven beans” on page 422.

  Applications can use message-driven beans as asynchronous message consumers. You deploy a message-driven bean as a message listener for a destination. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination.

Asynchronous beans - WebSphere Trader sample application

The Asynchronous beans - WebSphere Trader sample application illustrates how to implement a streaming stock ticker server and client using asynchronous beans and Java Platform, Enterprise Edition (Java EE) services such as:

- Servlets
- Java Message Service (JMS)
- Session enterprise beans
- Container-managed persistence (CMP) 2.0 enterprise beans
- Message-driven beans (MDB)

This sample uses several parts to maximize the utilization of a server:

- Work - Runs Java EE context-aware code on a thread.
- Alarm - Runs Java EE context-aware code at a given time interval.
- EventSource - A method of broadcasting events to registered listeners.
- SubsystemMonitor - A thread that monitors the status of any asynchronous system and uses an EventSource method to inform registered listeners of the system status.
- WorkManager - Thread configuration and Java EE context policies that are used by various asynchronous beans parts.
- AsynchScope - A collection of alarms, subsystem monitors and other asynchronous scopes that support relationships. This collection uses a single WorkManager thread and is also an event source.
- Startup Bean - A specialized, stateful session enterprise bean that supports bootstrapping asynchronous work when the application starts.

This sample is available from the Samples section of the information center.

Programming to use JMS and messaging directly

Your enterprise applications can use Java Message Service (JMS) programming interfaces directly to provide messaging services, and methods that implement business logic.

About this task

WebSphere Application Server supports asynchronous messaging as a method of communication based on JMS programming interfaces. Using JMS, enterprise applications can exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). An enterprise application can explicitly poll for messages on a destination.

If you choose not to use JNDI to obtain configuration information for your messaging provider, for example for connection factories or destinations, you can instead use an API provided by your messaging provider to specify that configuration information programmatically.

If you want to transmit messages between JMS applications and traditional WebSphere MQ applications, you must consider how the JMS message structure is mapped onto a WebSphere MQ message. This includes scenarios where you want to use WebSphere MQ to manipulate messages transmitted between two JMS applications; for example, by using WebSphere MQ as a message broker.

By default, JMS messages held on WebSphere MQ queues use an MQRFH2 header to hold some of the JMS message header information. Many traditional WebSphere MQ applications cannot process messages with these headers, and require their own characteristic headers, for example the MQWIH for WebSphere MQ Workflow applications. For more information about how the JMS message structure is mapped onto a WebSphere MQ message, see the section *Mapping JMS messages* in the WebSphere MQ information center.

Example

This following example shows how to programmatically configure a resource for the default messaging provider.

In this example, a JMS connection to a service integration bus is created by using the API in the com.ibm.websphere.sib package. This is an alternative to using JNDI to look up administratively configured connection factories. After the connection is established, the sample program reads lines of input from the console and sends them as JMS text messages to the specified destination.

This example can be run as a thin client application, or as a stand-alone client application.

```
/*
 * Sample program
 * © COPYRIGHT International Business Machines Corp. 2009
 * All Rights Reserved * Licensed Materials - Property of IBM
 *
 * This sample program is provided AS IS and may be used, executed,
 * copied and modified without royalty payment by customer
 *
 * (a) for its own instruction and study,
 * (b) in order to develop applications designed to run with an IBM
 *     WebSphere product for the customer's own internal use.
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import com.ibm.websphere.sib.api.jms.JmsConnectionFactory;
import com.ibm.websphere.sib.api.jms.JmsFactoryFactory;
import com.ibm.websphere.sib.api.jms.JmsQueue;
import com.ibm.websphere.sib.api.jms.JmsTopic;

/**
 * Sample code to programmatically create a connection to a bus and
 * send a text message.
 *
 * Example command lines:
 *   SIBusSender topic://my/topic?topicSpace=Default.Topic.Space MyBus localhost:7276
 *   SIBusSender queue://myQueue MyBus localhost:7286:BootstrapSecureMessaging InboundSecureMessaging
 */
public class SIBusSender {

  /**
   * @param args DEST_URL,BUS_NAME,PROVIDER_ENDPOINTS,[TRANSPORT_CHAIN]
   */
  public static void main(String[] args) throws JMSException, IOException {

    // Parse the arguments
```

```
if (args.length < 3) {
  throw new IllegalArgumentException(
      "Usage: SIBusSender <DEST_URL> <BUS_NAME> <PROVIDER_ENDPOINTS> [TARGET_TRANSPORT_CHAIN]");
}
String destUrl = args[0];
String busName = args[1];
String providerEndpoints = args[2];
String targetTransportChain = "InboundBasicMessaging";
if (args.length >= 4) targetTransportChain = args[3];

// Obtain the factory factory
JmsFactoryFactory jmsFact = JmsFactoryFactory.getInstance();

// Create a JMS destination
Destination dest;
if (destUrl.startsWith("topic://")) {
  JmsTopic topic = jmsFact.createTopic(destUrl);
  // Setter methods could be called here to configure the topic
  dest = topic ;
}
else {
  JmsQueue queue = jmsFact.createQueue(destUrl);
  // Setter methods could be called here to configure the queue
  dest = queue;
}

// Create a unified JMS connection factory
JmsConnectionFactory connFact = jmsFact.createConnectionFactory();

// Configure the connection factory
connFact.setBusName(busName);
connFact.setProviderEndpoints(providerEndpoints);
connFact.setTargetTransportChain(targetTransportChain);

// Create the connection
Connection conn = connFact.createConnection();

Session session = null;
MessageProducer producer = null;
try {

  // Create a session
  session = conn.createSession(false, // Not transactional
                               Session.AUTO_ACKNOWLEDGE);

  // Create a message producer
  producer = session.createProducer(dest);

  // Loop reading lines of text from the console to send
  System.out.println("Ready to send to " + dest + " on bus " + busName);
  BufferedReader lineInput = new BufferedReader(new InputStreamReader(System.in));
  String line = lineInput.readLine();
  while (line != null && line.length() > 0) {

    // Create a text message containing the line
    TextMessage message = session.createTextMessage();
    message.setText(line);

    // Send the message
    producer.send(message,
                  Message.DEFAULT_DELIVERY_MODE,
                  Message.DEFAULT_PRIORITY,
                  Message.DEFAULT_TIME_TO_LIVE);

    // Read the next line
    line = lineInput.readLine();
  }
```

```
    }
    // Finally block to ensure we close our JMS objects
    finally {

      // Close the message producer
      try {
        if (producer != null) producer.close();
      }
      catch (JMSException e) {
        System.err.println("Failed to close message producer: " + e);
      }

      // Close the session
      try {
        if (session != null) session.close();
      }
      catch (JMSException e) {
        System.err.println("Failed to close session: " + e);
      }

      // Close the connection
      try {
        conn.close();
      }
      catch (JMSException e) {
        System.err.println("Failed to close connection: " + e);
      }

    }
  }

}
```

Designing an enterprise application to use JMS

There are many things to consider when designing an enterprise application to use the JMS APIs directly for asynchronous messaging.

Procedure

- For messaging operations, you should write application programs that use only references to the interfaces defined in Sun's javax.jms package.

  JMS defines a generic view of messaging that maps onto the underlying transport. An enterprise application that uses JMS makes use of the following interfaces that are defined in Sun's javax.jms package:

  Connection
  : Provides access to the underlying transport, and is used to create Sessions.

  Session
  : Provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers.

  MessageProducer
  : Used to send messages.

  MessageConsumer
  : Used to receive messages.

  The generic JMS interfaces are subclassed into the following more specific versions for point-to-point and publish/subscribe behavior.

Table 76. The point-to-point and publish/subscribe versions of JMS common interfaces. The first column of this table lists the JMS common interfaces, the second column lists the corresponding point-to-point interfaces, and the third column lists the corresponding publish/subscribe interfaces.

JMS common interfaces	Point-to-point interfaces	Publish/subscribe interfaces
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession,	TopicSession,
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

For more information about using these JMS interfaces, see the Java Message Service Documentation and the *Using Java* section of the WebSphere MQ information center.

The section "Java Message Service (JMS) Requirements" of the J2EE specification gives a list of methods that must not be called in web and EJB containers:

```
javax.jms.Session method setMessageListener
javax.jms.Session method getMessageListener
javax.jms.Session method run
javax.jms.QueueConnection method createConnectionConsumer
javax.jms.TopicConnection method createConnectionConsumer
javax.jms.TopicConnection method createDurableConnectionConsumer
javax.jms.MessageConsumer method getMessageListener
javax.jms.MessageConsumer method setMessageListener
javax.jms.Connection setExceptionListener
javax.jms.Connection stop
javax.jms.Connection setClientID
```

This method restriction is enforced in WebSphere Application Server by throwing a javax.jms.IllegalStateException exception.

- Applications refer to JMS resources that are predefined, as administered objects, to WebSphere Application Server.

  Details of JMS resources that are used by enterprise applications are defined to WebSphere Application Server and bound into the JNDI namespace by the WebSphere administrative support. An enterprise application can retrieve these objects from the JNDI namespace and use them without needing to know anything about their implementation. This enables the underlying messaging architecture defined by the JMS resources to be changed without requiring changes to the enterprise application.

Table 77. JMS resources for point-to-point and publish/subscribe messaging. The left hand column of this table lists the JMS resources for point-to-point messaging, and the right hand column lists the JMS resources for publish/subscribe.

Point-to-point	Publish/subscribe
ConnectionFactory (or QueueConnectionFactory) Queue	ConnectionFactory (or TopicConnectionFactory) Topic

A connection factory is used to create connections from the JMS provider to the messaging system, and encapsulates the configuration parameters needed to create connections.

- To improve performance, the application server pools connections and sessions with the JMS provider. You have to configure the connection and session pool properties appropriately for your applications, otherwise you might not get the connection and session behavior that you want.
- Applications must not cache JMS connections, sessions, producers or consumers. WebSphere Application Server closes these objects when a bean or servlet completes, and so any attempt to use a cached object will fail with a javax.jms.IllegalStateException exception.

  To improve performance, applications can cache JMS objects that have been looked up from JNDI. For example, an EJB or servlet needs to look up a JMS ConnectionFactory only once, but it must call the

createConnection method on each instantiation. Because of the effect of pooling on connections and sessions with the JMS provider, there should be no performance impact.

- A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created.
- Using durable subscriptions with the default messaging provider. A durable subscription on a JMS topic enables a subscriber to receive a copy of all messages published to that topic, even after periods of time when the subscriber is not connected to the server. Therefore, subscriber applications can operate disconnected from the server for long periods of time, and then reconnect to the server and process messages that were published during their absence. If an application creates a durable subscription, it is added to the runtime list that administrators can display and act on through the administrative console.

  Each durable subscription is given a unique identifier, *clientID##subName* where:

  clientID
  : The client identifier used to associate a connection and its objects with the messages maintained for applications (as clients of the JMS provider). You should use a naming convention that helps you identify the applications, in case you have to relate durable subscriptions to the associated applications for runtime administration.

  subName
  : The subscription name used to uniquely identify a durable subscription within a given client identifier.

  For durable subscriptions created by message-driven beans, these values are set on the JMS activation specification. For other durable subscriptions, the client identifier is set on the JMS connection factory, and the subscription name is set by the application on the createDurableSubscriber operation.

  To create a durable subscription to a topic, an application uses the createDurableSubscriber operation defined in the JMS API:

```
public TopicSubscriber createDurableSubscriber(Topic topic,
                          java.lang.String subName,
                          java.lang.String messageSelector,
                          boolean noLocal)
                          throws JMSException
```

  topic The name of the JMS topic to subscribe to. This is the name of an object supporting the javax.jms.Topic interfaces, such as found by looking up a suitable JNDI entry.

  subName
  : The name used to identify this subscription.

  messageSelector
  : Only messages with properties matching the message selector expression are delivered to consumers. A value of null or an empty string indicates that all messages should be delivered.

  noLocal
  : If set to `true`, this parameter prevents the delivery of messages published on the same connection as the durable subscriber.

  Applications can use a two argument form of the createDurableSubscriber operation that takes only topic and subName parameters. This alternative call directly invokes the four argument version shown above, but sets messageSelector to null (so all messages are delivered) and sets noLocal to false (so messages published on the connection are delivered). For example, to create a durable subscription to the topic called `myTopic`, with the subscription name of `mySubscription`:

```
session.createDurableSubscriber(myTopic,"mySubscription");
```

  If the createDurableSubscription operation fails, it throws a JMS exception that provides a message and linked exception to give more detail about the cause of the problem.

  To delete a durable subscription, an application uses the unsubscribe operation defined in the JMS API

  In normal operation there can be at most one active (connected) subscriber for a durable subscription at a time. However, the subscriber application can be running in a cloned application server, for failover

and load balancing purposes. In this case the “one active subscriber” restriction is lifted to provide a shared durable subscription that can have multiple simultaneous consumers.

For more information about application use of durable subscriptions, see the section “Using Durable Subscriptions” in the JMS specification.

- Decide what message selectors are needed. You can use the JMS message selector mechanism to select a subset of the messages on a queue so that this subset is returned by a receive call. The selector can refer to fields in the JMS message header and fields in the message properties.
- Acting on messages received. When a message is received, you can act on it as needed by the business logic of the application. Some general JMS actions are to check that the message is of the correct type and extract the content of the message. To extract the content from the body of the message, cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as TextMessage. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

  In this example, the instanceof operator is used to check that the message received is of the `TextMessage` type. The message content is then extracted by casting to the TextMessage subclass.

```
       if ( inMessage instanceof TextMessage )

...
            String replyString = ((TextMessage) inMessage).getText();
```

- JMS applications using the default messaging provider can access, without any restrictions, the content of messages that have been received from WebSphere Application Server Version 5 embedded messaging or WebSphere MQ.
- JMS applications can access the full set of JMS_IBM* properties. These properties are of value to JMS applications that use resources provided by the default messaging provider, the V5 default messaging provider, or the WebSphere MQ provider.

  For messages handled by WebSphere MQ, the JMS_IBM* properties are mapped to equivalent WebSphere MQ Message Descriptor (MQMD) fields. For more information about the JMS_IBM* properties and MQMD fields, see the *Using Java* section of the WebSphere MQ information center.
- JMS applications can use report messages as a form of managed request/response processing, to give remote feedback to producers on the outcome of their send operations and the fate of their messages. JMS applications can request a full range of report options using **`JMS_IBM_Report_Xxxx`** message properties. For more information about using JMS report messages, see “JMS report messages” on page 621.
- JMS applications can use the **`JMS_IBM_Report_Discard_Msg`** property to control how a request message is disposed of if it cannot be delivered to the destination queue.

  MQRO_Dead_Letter_Queue
  This is the default. The request message should be written to the dead letter queue.

  MQRO_Discard
  The request message should be discarded. This is usually used in conjunction with MQRO_Exception_With_Full_Data to return an undeliverable request message to its sender.
- Using a listener to receive messages asynchronously. In a client, not in a servlet or enterprise bean, an alternative to making calls to QueueReceiver.receive() is to register a method that is called automatically when a suitable message is available; for example:

```
...
MyClass listener =new MyClass();
queueReceiver.setMessageListener(listener);
//application continues with other application-specific behavior.
...
```

  When a message is available, it is retrieved by the onMessage() method on the listener object.

```
import javax.jms.*;
public class MyClass implements MessageListener
{
public void onMessage(Message message)
```

```
{
System.out.println("message is "+message);
//application specific processing here
...

}
}
```

For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to receive() methods. To cope with this situation, you can register an ExceptionListener, which is an instance of a class that implements the onException() method. When an error occurs, this method is called with the **JMSException** passed as its only parameter.

For more details about using listeners to receive messages asynchronously, see the Java Message Service Documentation.

Note: An alternative to developing your own JMS listener class, you can use a message-driven bean, as described in Programming with message-driven beans.

- Take care when performing a JMS receive() from a server-side application component if that receive() invocation is waiting on a message produced by another application component that is deployed in the same server. Such a JMS receive() is synchronous, so blocks until the response message is received.

  This type of application design can lead to the consumer or producer problem where the entire set of work threads can be exhausted by the receiving component, which has been blocked waiting for responses, leaving no available worker thread for which to dispatch the application component that would generate the response JMS message. For example, a servlet and a message-driven bean are deployed in the same server. When this servlet dispatches a request it sends a message to a queue that is serviced by the message-driven bean (that is, messages produced by the servlet are consumed by the message-driven bean onMessage() method). The servlet subsequently issues a receive(), waiting for a reply on a temporary ReplyTo queue. The message-driven bean onMessage() method performs a database query and sends back a reply to the servlet on the temporary queue. If a large number of servlet requests occur at once (relative to the number of server worker threads), then all available server worker threads will probably be used to dispatch a servlet request, send a message, and wait for a reply. The application server then enters a condition where no threads remain to process any of the message-driven beans that are now pending. Because the servlets are waiting in blocking receives, the server hangs, likely leading to application failure.

  Possible solutions are:

  1. Ensure that the number of worker threads (# of threads per server region * # of server regions per server) exceeds the number of concurrent dispatches of the application component doing the receive() so that there is always a worker thread available to dispatch the message producing component.
  2. Use an application topology that places the receiver application component in a separate server from the producer application component. Although worker thread usage might still have to be carefully considered under such a deployment scenario, this separation ensures that there are always be threads that cannot be blocked by the message receiving component. There can be other interactions to consider, such as an application server that has multiple applications installed.
  3. Refactor your application to do the message receives from a client component, which will not compete with the producer component for worker threads. Furthermore, the client component can do asynchronous (non-blocking) receives, which are prohibited from J2EE servers. So, for example, the example application above can be refactored to have a client sending messages to a queue and then waiting for a response from the MDB.

- If you want to use authentication with WebSphere MQ or the Version 5 Embedded Messaging support, you cannot have user IDs longer than 12 characters. For example, the default Windows NT user ID, **administrator**, is not valid for use with WebSphere internal messaging, because it contains 13 characters.
- The following points, as defined in the EJB specification, apply to the use of flags on create*xxx*Session calls:

  - The transacted flag passed on create*xxx*Session is ignored inside a global transaction and all work is performed as part of the transaction. Outside of a transaction the transacted flag is used and, if set to `true`, the application should use session.commit() and session.rollback() to control the completion of the work. In an EJB2.0 module, if the transacted flag is set to `true` and outside of an XA transaction, then the session is involved in the WebSphere local transaction and the unresolved action attribute of the method applies to the JMS work if it is not committed or rolled back by the application.
  - Clients cannot use Message.acknowledge() to acknowledge messages. If a value of `CLIENT_ACKNOWLEDGE` is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.
- If you want your application to use WebSphere MQ as an external JMS provider, then send messages within a container-managed transaction.

  When you use WebSphere MQ as an external JMS provider, messages sent within a user-managed transaction can arrive before the transaction commits. This occurs only when you use WebSphere MQ as an external JMS provider, and you send messages to a WebSphere MQ queue within a user-managed transaction. The message arrives on the destination queue before the transaction commits.

  The cause of this problem is that the WebSphere MQ resource manager has not been enlisted in the user-managed transaction.

  The solution is to use a container-managed transaction.

Developing an enterprise application to use JMS

Use this task to develop an enterprise application to use the JMS API directly for asynchronous messaging.

About this task

This topic gives an overview of the steps needed to develop an enterprise application (servlet or enterprise bean) to use the JMS API directly for asynchronous messaging.

This topic only describes the JMS-related case; it does not describe general enterprise application programming, which you should already be familiar with. For detailed information about these steps, and for examples of developing an enterprise application to use JMS, see the Java Message Service Documentation

Details of JMS resources that are used by enterprise applications are defined to WebSphere Application Server and bound into the JNDI namespace by the WebSphere Application Server administrative support.

To use JMS, complete the following general steps:

Procedure

1. Import JMS packages. An enterprise application that uses JMS starts with a number of import statements for JMS, which should include at least the following statements:

   ```
   import javax.jms.*;       //JMS interfaces
   import javax.naming.*;    //Used for JNDI lookup of administered objects
   ```

2. Get an initial context:

   ```
      try     {
           ctx = new InitialContext(env);
   ...
   ```

3. Retrieve administered objects from the JNDI namespace. The InitialContext.lookup() method is used to retrieve administered objects (a JMS connection factory and JMS destinations). The following example shows how to receive a message from a queue:

```
        qcf = (QueueConnectionFactory)ctx.lookup( qcfName );
...
        inQueue = (Queue)ctx.lookup( qnameIn );
...
```

An alternative, but less manageable, approach to obtaining administratively-defined JMS destination objects by JNDI lookup is to use the Session.createQueue(String) method or Session.createTopic(String) method. For example:

```
Queue q = mySession.createQueue("Q1");
```

creates a JMS Queue instance that can be used to reference the existing destination Q1.

In its simplest form, the parameter to these create methods is the name of an existing destination. For more complex situations, applications can use a URI-based format, which allows an arbitrary number of name value pairs to be supplied to set various properties of the JMS destination object.

4. Create a connection to the messaging service provider. The connection provides access to the underlying transport, and is used to create sessions. The createQueueConnection() method on the factory object is used to create the connection.

   ```
   connection = qcf.createQueueConnection();
   ```

   The JMS specification defines that connections should be created in the stopped state. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. To start the connection, issue the following command:

   ```
   connection.start();
   ```

5. Create a session, for sending or receiving messages. The session provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers. The createQueueSession method is used on the connection to obtain a session. The method takes two parameters:
   - A boolean that determines whether the session is transacted.
   - A parameter that determines the acknowledge mode.

   ```
   boolean transacted = false;
   session = connection.createQueueSession( transacted,
                                            Session.AUTO_ACKNOWLEDGE);
   ```

   In this example, the session is not transacted, and it should automatically acknowledge received messages. With these settings, a message is backed out only after a system error or if the application terminates unexpectedly.

   The following points, as defined in the EJB specification, apply to these flags:
   - The transacted flag passed on createQueueSession is ignored inside a global transaction and all work is performed as part of the transaction. Outside of a transaction the transacted flag is used and, if set to true, the application should use session.commit() and session.rollback() to control the completion of the work. In an EJB2.0 module, if the transacted flag is set to `true` and outside of an XA transaction, then the session is involved in the WebSphere local transaction and the unresolved action attribute of the method applies to the JMS work if it is not committed or rolled back by the application.
   - Clients cannot use Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

6. Send a message.
   a. Create MessageProducers to create messages. For point-to-point messaging the MessageProducer is a QueueSender that is created by passing an output queue object (retrieved earlier) into the createSender method on the session. A QueueSender is usually created for a specific queue, so that all messages sent by that sender are sent to the same destination.

      ```
      QueueSender queueSender = session.createSender(inQueue);
      ```

   b. Create the message. Use the session to create an empty message and add the data passed.

JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the vendor-specific class names for the message types, methods are provided on the Session object for message creation.

In this example, a text message is created from the outString property:

```
TextMessage outMessage = session.createTextMessage(outString);
```

c. Send the message.

To send the message, the message is passed to the send method on the QueueSender:

```
queueSender.send(outMessage);
```

7. Receive replies.

a. Create a correlation ID to link the message sent with any replies. In this example, the client receives reply messages that are related to the message that it has sent, by using a provider-specific message ID in a JMSCorrelationID.

```
messageID = outMessage.getJMSMessageID();
```

The correlation ID is then used in a message selector, to select only messages that have that ID:

```
String selector = "JMSCorrelationID = '"+messageID+"'";
```

b. Create a MessageReceiver to receive messages. For point-to-point the MessageReceiver is a QueueReceiver that is created by passing an input queue object (retrieved earlier) and the message selector into the createReceiver method on the session.

```
QueueReceiver queueReceiver = session.createReceiver(outQueue, selector);
```

c. Retrieve the reply message. To retrieve a reply message, the receive method on the QueueReceiver is used:

```
Message inMessage = queueReceiver.receive(2000);
```

The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. If you omit this parameter, the call blocks indefinitely. If you do not want any delay, use the receiveNoWait()method. In this example, the receive call returns when the message arrives, or after 2000ms, whichever is sooner.

d. Act on the message received. When a message is received, you can act on it as needed by the business logic of the client. Some general JMS actions are to check that the message is of the correct type and extract the content of the message. To extract the content from the body of the message, it is necessary to cast from the generic Message class (which is the declared return type of the receive methods) to the more specific subclass, such as TextMessage. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

In this example, the instanceof operator is used to check that the message received is of the TextMessage type. The message content is then extracted by casting to the TextMessage subclass.

```
if ( inMessage instanceof TextMessage )

...
        String replyString = ((TextMessage) inMessage).getText();
```

8. Closing down. If the application needs to create many short-lived JMS objects at the Session level or lower, it is important to close all the JMS resources used. To do this, you call the close() method on the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.

```
queueReceiver.close();
...
queueSender.close();
...
session.close();
```

```
        session = null;
...
        connection.close();
        connection = null;
```

9. Publishing and subscribing to messages. To use JMS Publish/Subscribe support instead of point-to-point messaging, the general actions are the same; for example, to create a session and connection. The exceptions are that topic resources are used instead of queue resources (such as TopicPublisher instead of QueueSender), as shown in the following example to publish a message:

```
// Creating a TopicPublisher
        TopicPublisher pub = session.createPublisher(topic);
...
        pub.publish(outMessage);
...
        // Closing TopicPublisher
        pub.close();
```

10. Handling errors Any JMS runtime errors are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

    Unlike normal Java exceptions, a JMSException can contain another exception embedded in it. The implementation of JMSException does not include the embedded exception in the output of its toString()method. Therefore, you must check explicitly for an embedded exception and print it out, as shown in the following example:

```
catch (JMSException je)
{
   System.out.println("JMS failed with "+je);
   Exception le = je.getLinkedException();
   if (le != null)
   {
       System.out.println("linked exception "+le);
   }
}
```

What to do next

After you have packaged your application, you can next deploy the application into WebSphere Application Server, as described in “Deploying an enterprise application to use JMS” on page 2083.

Developing a JMS client

Use this task to develop a JMS client application to use messages to communicate with enterprise applications.

About this task

This topic gives an overview of the steps needed to develop a JMS client application. This topic only describes the JMS-related case; it does not describe general client programming, which you should already be familiar with. For detailed information about these steps, and for examples of developing JMS clients, see the Java Message Service Documentation and the *Using Java* section of the WebSphere MQ information center.

A JMS client assumes that the JMS resources (such as a queue connection factory and queue destination) already exist. A client application can obtain suitable JMS resources either by JNDI lookup or programmatically without using JNDI.

For information about the Thin Client for JMS with WebSphere Application Server, which is an embeddable technology that provides JMS V1.1 connections to a WebSphere Application Server default messaging provider messaging engine, see Using JMS to connect to a WebSphere Application Server default messaging provider messaging engine.

For more information about developing client applications and configuring JMS resources for them, see Developing J2EE application client code and related tasks.

To use JMS, a typical JMS client program completes the following general steps. This example is based on the use of JNDI lookups to obtain JMS resources.

Procedure

1. Import JMS packages. An enterprise application that uses JMS starts with a number of import statements for JMS; for example:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.jms.*;
```

2. Get an initial context.

```
   try     {
        ctx = new InitialContext(env);
...
```

3. Define the parameters that the client is to use; for example, to identify the queue connection factory and to assemble a message to be sent.

```
public class JMSppSampleClient
{
  public static void main(String[] args)
  throws JMSException, Exception

  {
    String  messageID                = null;
    String  outString                = null;
    String  qcfName                  = "java:comp/env/jms/ConnectionFactory";
    String  qnameIn                  = "java:comp/env/jms/Q1";
    String  qnameOut                 = "java:comp/env/jms/Q2";
    boolean verbose                  = false;

    QueueSession           session    = null;
    QueueConnection        connection = null;
    Context                ctx        = null;

    QueueConnectionFactory qcf        = null;
    Queue                  inQueue    = null;
    Queue                  outQueue   = null;

...
```

4. Retrieve administered objects from the JNDI namespace. The InitialContext.lookup() method is used to retrieve administered objects (a queue connection factory and the queue destinations):

```
       qcf = (QueueConnectionFactory)ctx.lookup( qcfName );
...
       inQueue = (Queue)ctx.lookup( qnameIn );
       outQueue = (Queue)ctx.lookup( qnameOut );
...
```

5. Create a connection to the messaging service provider. The connection provides access to the underlying transport, and is used to create sessions. The createQueueConnection() method on the factory object is used to create the connection.

```
       connection = qcf.createQueueConnection();
```

   The JMS specification defines that connections should be created in the stopped state. Until the connection starts, MessageConsumers that are associated with the connection cannot receive any messages. To start the connection, issue the following command:

```
       connection.start();
```

6. Create a session, for sending and receiving messages. The session provides a context for producing and consuming messages, including the methods used to create MessageProducers and MessageConsumers. The createQueueSession method is used on the connection to obtain a session. The method takes two parameters:
   - A boolean that determines whether the session is transacted.
   - A parameter that determines the acknowledge mode.

   ```
   boolean transacted = false;
   session = connection.createQueueSession( transacted,
                                            Session.AUTO_ACKNOWLEDGE);
   ```

   In this example, the session is not transacted, and it should automatically acknowledge received messages. With these settings, a message is backed out only after a system error or if the client application terminates unexpectedly.
7. Send the message.
   a. Create MessageProducers to create messages. For point-to-point the MessageProducer is a QueueSender that is created by passing an output queue object (retrieved earlier) into the createSender method on the session. A QueueSender is usually created for a specific queue, so that all messages sent by that sender are sent to the same destination.

      ```
      QueueSender queueSender = session.createSender(inQueue);
      ```

   b. Create the message. Use the session to create an empty message and add the data passed.

      JMS provides several message types, each of which embodies some knowledge of its content. To avoid referencing the vendor-specific class names for the message types, methods are provided on the Session object for message creation.

      In this example, a text message is created from the outString property, which could be provided as an input parameter on invocation of the client program or constructed in some other way:

      ```
      TextMessage outMessage = session.createTextMessage(outString);
      ```

   c. Send the message.

      To send the message, the message is passed to the send method on the QueueSender:

      ```
      queueSender.send(outMessage);
      ```

8. Receive replies.
   a. Create a correlation ID to link the message sent with any replies. In this example, the client receives reply messages that are related to the message that it has sent, by using a provider-specific message ID in a JMSCorrelationID.

      ```
      messageID = outMessage.getJMSMessageID();
      ```

      The correlation ID is then used in a message selector, to select only messages that have that ID:

      ```
      String selector = "JMSCorrelationID = '"+messageID+"'";
      ```

   b. Create a MessageReceiver to receive messages. For point-to-point the MessageReceiver is a QueueReceiver that is created by passing an input queue object (retrieved earlier) and the message selector into the createReceiver method on the session.

      ```
      QueueReceiver queueReceiver = session.createReceiver(outQueue, selector);
      ```

   c. Retrieve the reply message. To retrieve a reply message, the receive method on the QueueReceiver is used:

      ```
      Message inMessage = queueReceiver.receive(2000);
      ```

      The parameter in the receive call is a timeout in milliseconds. This parameter defines how long the method should wait if there is no message available immediately. If you omit this parameter, the call blocks indefinitely. If you do not want any delay, use the receiveNoWait()method. In this example, the receive call returns when the message arrives, or after 2000ms, whichever is sooner.
   d. Act on the message received. When a message is received, you can act on it as needed by the business logic of the client. Some general JMS actions are to check that the message is of the correct type and extract the content of the message. To extract the content from the body of the message, cast from the generic Message class (which is the declared return type of the receive

methods) to the more specific subclass, such as TextMessage. It is good practice always to test the message class before casting, so that unexpected errors can be handled gracefully.

In this example, the instanceof operator is used to check that the message received is of the TextMessage type. The message content is then extracted by casting to the TextMessage subclass.

```
       if ( inMessage instanceof TextMessage )

...
            String replyString = ((TextMessage) inMessage).getText();
```

9. Closing down. If the application needs to create many short-lived JMS objects at the Session level or lower, it is important to close all the JMS resources used. To do this, you call the close() method on the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.

```
       queueReceiver.close();
...
       queueSender.close();
...
       session.close();
       session = null;
...
       connection.close();
       connection = null;
```

10. Publishing and subscribing messages. To use publish/subscribe support instead of point-to-point messaging, the general client actions are the same; for example, to create a session and connection. The exceptions are that topic resources are used instead of queue resources (such as TopicPublisher instead of QueueSender), as shown in the following example to publish a message:

```
// Creating a TopicPublisher
       TopicPublisher pub = session.createPublisher(topic);
...
       pub.publish(outMessage);
...
       // Closing TopicPublisher
       pub.close();
```

11. Handling errors Any JMS runtime errors are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and display them on a suitable output.

    Unlike normal Java exceptions, a JMSException can contain another exception embedded in it. The implementation of JMSException does not include the embedded exception in the output of its toString()method. Therefore, you must check explicitly for an embedded exception and print it out, as shown in the following example:

```
catch (JMSException je)
{
   System.out.println("JMS failed with "+je);
   Exception le = je.getLinkedException();
   if (le != null)
   {
       System.out.println("linked exception "+le);
   }
}
```

What to do next

For information about running a client against a specific remote server: “Running a Java EE client application with launchClient” on page 2015.

Programming for interoperation with WebSphere MQ

There are some differences between the WebSphere Application Server environment and the WebSphere MQ environment. If you are writing messaging programs that interoperate between these two environments, you should be aware of these differences and take them into account when designing, coding and deploying your programs.

Procedure

1. Learn more about the environment differences and other relevant concepts in How messages are passed between service integration and a WebSphere MQ network.
2. Read about designing programs that interoperate with WebSphere MQ in "Designing an application for interoperation with WebSphere MQ."

Designing an application for interoperation with WebSphere MQ

To design an application to interoperate with queue managers in a WebSphere MQ network you need to first consider the differences between the two environments, then design your JMS client based on the Java EE pattern, then identify any name-handling incompatibilities between the service integration bus and WebSphere MQ environments, then define the topic mappings.

Before you begin

Identify the WebSphere MQ queues with which your applications will interoperate. The exact names and locations can be left to the installation.

Procedure

1. Familiarize yourself with important reference information for the two interoperating environments, WebSphere MQ and the service integration bus.

   There are three types of reference material:

   - For mapping that is unique to service integration bus messaging, see "Mapping additional MQRFH2 header fields in service integration" on page 607.
   - For mapping between WebSphere Application Server service integration bus messaging and WebSphere MQ, see How service integration converts messages to and from WebSphere MQ format and "Mapping additional MQRFH2 header fields in service integration" on page 607.
   - For the differences between the WebSphere MQ functions and the service integration bus, see "WebSphere MQ functions not supported by service integration" on page 611.

2. Design your JMS client based on the typical Java EE pattern:
   a. Use JNDI to find a ConnectionFactory object.
   b. Use JNDI to find one or more Destination objects.
   c. Use the connection factory to create a JMS Connection object.
   d. Use the JMS connection to create one or more JMS Session objects.
   e. Use a JMS session and the destinations to create the MessageProducer and MessageConsumer objects.
   f. Start delivery of messages by starting the JMS connection.

   At this point a client has the basic JMS setup needed to produce and consume messages.
3. Identify any name-handling incompatibilities between the service integration bus and WebSphere MQ environments. If necessary, identify alias requirements, so that the WebSphere MQ application can handle service integration bus destination names of greater than 48 characters. For more information, see How to address bus destinations and WebSphere MQ queues.
4. Identify any reply destinations that are used by your application and check them for name-handling incompatibilities. For more information, see "Mapping destinations to and from WebSphere MQ queues, topics, and destinations" on page 601.

5. If your application publishes messages that you want to be forwarded to WebSphere MQ brokers, work with your administrator to define appropriate topic mappings on a publish/subscribe broker profile. You must also define topic mappings for any permanent reply topics. See Reply-to topics for request-reply messaging through a WebSphere MQ link and Request-reply messaging through a WebSphere MQ link for more information.

Mapping the message body to and from WebSphere MQ format

The WebSphere MQ message header (MQRFH2) and descriptor (MQMD) can contain information about the format of the WebSphere MQ message body. Service integration uses information contained in the MQRFH2 and MQMD when converting a message from WebSphere MQ format, and puts information into the MQRFH2 and MQMD when converting a message to WebSphere MQ format.

Exchanging messages between JMS programs through service integration and WebSphere MQ

Usually, you do not have to be aware of conversion between message formats to exchange JMS messages between service integration and WebSphere MQ, because service integration performs the appropriate conversion automatically, including character and numeric encoding. However, you might have to learn about message conversion if your JMS applications do not behave as expected, or if your service integration configuration includes JMS programs or mediations that process messages to or from non-JMS WebSphere MQ programs.

If your service integration applications exchange MapMessage objects with WebSphere MQ applications, you might have to specify a non-default map message encoding format.

WebSphere MQ message payload: format indications

The WebSphere MQ format message contains the following two indications of the payload format:

MQRFH2 <mcd> folder, Msd field

This field can contain information about the payload format. This is the “JMS format” information.

- When service integration converts a message to WebSphere MQ format, it automatically sets the appropriate value for the JMS message class.
- When service integration converts a message from WebSphere MQ format, it uses the value in this field (if there is an MQRFH2 that contains the field) to set the JMS message class.

JMS message class	MQRFH2 <mcd> folder, Msd field (“JMS format”)
TextMessage	jms_text
BytesMessage	jms_bytes
StreamMessage	jms_stream
MapMessage	jms_map
ObjectMessage	jms_object
Message	jms_none

If the “JMS format” information is not available, for example if there is no MQRFH2, service integration sets the JMS message class based on the “MQ format”.

For more information about the MQRFH2 <mcd> folder, see the WebSphere MQ Using Java documentation.

MQRFH2 (or MQMD) format field

The MQRFH2 (or the MQMD if there is no MQRFH2) format field contains information about the payload format. This is the “MQ format” information. Typically it contains MQFMT_STRING, which indicates that the payload is character data (and can be translated to a different codepage by

WebSphere MQ), or MQFMT_NONE, which indicates that the payload is not character data. These values are suitable for most JMS messages, and when service integration converts a message to WebSphere MQ format it automatically sets this field to one of the following values:

JMS message class	MQRFH2 (or MQMD) format field ("MQ format")
TextMessage	MQFMT_STRING
BytesMessage	MQFMT_NONE
StreamMessage	MQFMT_STRING
MapMessage	MQFMT_STRING
ObjectMessage	MQFMT_NONE
Message	MQFMT_NONE

If your application constructs messages for a WebSphere MQ application that requires a different format value, you can override the value from the previous table by setting the JMS_IBM_Format property to the required value. A particular example is when the WebSphere MQ application requires an additional header (for example, the MQCIH header for a CICS bridge application). Your application constructs a BytesMessage object that contains the header followed by any other message data, then replaces the default "MQ format" (MQFMT_NONE) by setting the JMS_IBM_Format property to the appropriate value for the header (for example, MQFMT_CICS for an MQCIH header).

When service integration converts a message from WebSphere MQ format, it sets the JMS_IBM_Format property to the value in the "MQ format" field. If the "JMS format" is not available, for example if there is no MQRFH2, service integration sets the JMS message class to TextMessage if the "MQ format" is MQFMT_STRING and to BytesMessage otherwise.

For more information about the MQRFH2 (or MQMD) format field, see the WebSphere MQ Application Programming Reference.

WebSphere MQ message payload: character and numeric encoding

In addition to the format field, the MQRFH2 (or the MQMD if there is no MQRFH2) contains fields that identify the character encoding and numeric encoding for the message payload.

When service integration converts a message to WebSphere MQ format, it automatically selects default values (UTF-8 character encoding and big-endian numeric encoding) that are suitable for most JMS messages. If your application constructs messages for a WebSphere MQ application that requires a different character or numeric encoding, you can override the character encoding value by setting the **JMS_IBM_Character_Set** property to the required coded character set ID (CCSID), or the **JMS_IBM_Encoding** property to the required numeric format, or both. For information about the values you can use for **JMS_IBM_Character_Set** and **JMS_IBM_Encoding**, see the documentation in the WebSphere MQ library.

When the JMS message has a body that is encoded as character data in WebSphere MQ (TextMessage, StreamMessage, or MapMessage), setting **JMS_IBM_Character_Set** causes service integration to convert the text to that coded character set in the WebSphere MQ message body.

When the JMS message has a body that is not character data (BytesMessage or ObjectMessage), setting **JMS_IBM_Character_Set** does not cause service integration to convert the bytes; it indicates to WebSphere MQ that any character data in the message body is already encoded using the specified coded character set. If the value of the **JMS_IBM_Format** is a format that WebSphere MQ recognises, it can convert that character data to the coded character set that the receiving application requires.

When service integration converts a message from WebSphere MQ format, it sets the **JMS_IBM_Character_Set** and **JMS_IBM_Encoding** properties from the fields in the MQRFH2 (or the MQMD if there is no MQRFH2). If the JMS message is a TextMessage, StreamMessage, MapMessage, or ObjectMessage, your application makes no use of the values of the **JMS_IBM_Character_Set** and

JMS_IBM_Encoding properties. If the JMS message is a BytesMessage, then the body of the JMS message is binary data. In this case, your application must be aware of the values of the **JMS_IBM_Character_Set** and **JMS_IBM_Encoding** properties, because they indicate the encoding of any character data or numeric data that is embedded within the binary data of the message.

Mapping the message header fields and properties to and from WebSphere MQ format

When service integration converts a message to WebSphere MQ format, it sets fields in the MQMD and the MQRFH2 based on the service integration message header fields and properties; these include JMS message header fields and properties applicable to the message. When service integration converts a message from WebSphere MQ format, it sets the service integration message header fields and properties from the MQMD and the MQRFH2 in the WebSphere MQ message.

Exchanging messages between JMS programs through service integration and WebSphere MQ

Usually, you do not have to be aware of conversion between message formats to exchange JMS messages between service integration and WebSphere MQ, because service integration performs the appropriate conversion automatically, including character and numeric encoding. However, you might have to learn about message conversion if your JMS applications do not behave as expected, or if your service integration configuration includes JMS programs or mediations that process messages to or from non-JMS WebSphere MQ programs.

If your service integration applications exchange MapMessage objects with WebSphere MQ applications, you might have to specify a non-default map message encoding format.

WebSphere MQ message properties: the MQMD and the MQRFH2

WebSphere MQ messages contain message properties in the message descriptor (MQMD) and in the rules and formatting header 2 (MQRFH2). The WebSphere MQ message always includes an MQMD, but the MQRFH2 is optional because some WebSphere MQ applications cannot process messages that contain an MQRFH2. To simplify interoperation, you can configure service integration to omit the MQRFH2 from messages for applications that cannot process the MQRFH2. When service integration omits the MQRFH2, it discards the corresponding service integration header fields and properties.

Note: A small amount of the MQRFH2 information is also stored in MQMD fields. However, these MQMD fields are not exact equivalents, tend to be less specific, and cannot be relied upon to provide an adequate substitute for the MQRFH2 information. Therefore if the receiving application can accept an MQRFH2 header, you should always provide one.

Similarly, service integration might receive messages from WebSphere MQ applications that generate messages with no MQRFH2. When service integration receives a message with no MQRFH2, it creates a “best guess” service integration header, by getting as much information as it can from the MQMD, and using default values for the other fields.

For detailed information about the contents of the message descriptor and the message headers, see the WebSphere MQ Application Programming Reference. For details of WebSphere MQ JMS support, including details of how WebSphere MQ stores JMS message properties and header fields in the MQMD and the MQRFH2, see WebSphere MQ Using Java.

WebSphere MQ message properties: JMS header fields

The following table shows how service integration maps JMS header fields to and from MQMD and MQRFH2 fields when converting messages to and from WebSphere MQ format.

The table shows the MQRFH2 field as *folder.field*, where *folder* is the name of the MQRFH2 folder that contains the field, and *field* is the name of the field within the MQRFH2 folder.

For several JMS header fields, there is both an MQMD field and an MQRFH2 field. When service integration is converting messages to WebSphere MQ format, it sets both the MQMD and the MQRFH2 fields. When service integration is converting messages from WebSphere MQ format, it sets the JMS header field from the MQRFH2 field if it is available, otherwise from the MQMD field.

Table 78. JMS, MQMD and MQRFH2 header fields. The first column of this table lists the JMS header fields, and the second column shows the MQMD fields that relate to the JMS header fields in the first column. The third column shows the MQRFH2 fields that relate to the JMS header fields in the first column. The fourth column provides links, where required, to the footnotes that appear after the table.

JMS header field	MQMD field	MQRFH2 field	Notes
JMSCorrelationID	**CorrelId**	**jms.Cid**	See Note 1.
JMSDeliveryMode	**Persistence**	**jms.Dlv**	See Note 15.
JMSDestination		**jms.Dst**	See Note 16.
JMSExpiration	**Expiry**	**jms.Exp**	
JMSMessageID	**MsgId**		
JMSPriority	**Priority**		See Note 2.
JMSRedelivered	**BackoutCount**		See Note 3.
JMSReplyTo	**ReplyToQ** and **ReplyToQMgr**	**jms.Rto**	See Note 16.
JMSTimestamp	**PutDate** and **PutTime**	**jms.Tms**	
JMSType		**mcd.Type**	

Note 1: The MQMD **CorrelId** field can hold a standard WebSphere MQ Correlation ID of 48 hexadecimal digits (24 bytes). The **JMSCorrelationID** can be a `byte[]` value, a string value containing hexadecimal characters and prefixed with “ID:”, or an arbitrary string value not beginning “ID:”. The first two of these represent a standard WebSphere MQ Correlation ID and map directly to or from the MQMD **CorrelId** field (truncated or padded with zeros as applicable); they do not use the MQRFH2 **jms.Cid** field. The third (arbitrary string) uses the MQRFH2 **jms.Cid** field; the first 24 bytes of the string, in UTF-8 format, are written into the MQMD **CorrelID**.

Note 2: WebSphere MQ stores the **JMSPriority** value in the MQRFH2 **jms.Pri** field but does not use any value already in that field. Service integration does not check or set the MQRFH2 **jms.Pri** field.

Note 3: Service integration sets the **JMSRedelivered** indicator for a message it receives from WebSphere MQ based on the **BackoutCount** field of the MQMD; a non-zero **BackoutCount** value indicates that a previous receive for the message was rolled back.

WebSphere MQ message properties: JMS defined properties

The following table shows how service integration maps JMS defined properties to and from MQMD and MQRFH2 fields when converting messages to and from WebSphere MQ format.

The table shows the MQRFH2 field as *folder.field*, where *folder* is the name of the MQRFH2 folder that contains the field, and *field* is the name of the field within the MQRFH2 folder.

For several JMS-defined properties, there is both an MQMD field and an MQRFH2 field. When service integration is converting messages to WebSphere MQ format, it sets both the MQMD and the MQRFH2 fields. When service integration is converting messages from WebSphere MQ format, it sets the JMS defined property from the MQRFH2 field if it is available, otherwise from the MQMD field.

Table 79. JMS properties with MQMD and MQRFH2 fields. The first column of this table lists the JMS defined properties, and the second column shows the MQMD fields that relate to the JMS defined properties in the first column. The third column shows the MQRFH2 fields that relate to the JMS defined properties in the first column. The fourth column provides links, where required, to the footnotes that appear after the table.

JMS defined property	MQMD field	MQRFH2 field	Notes
JMSXAppID	**PutApplName**		
JMSXDeliveryCount	**BackoutCount**		
JMSXGroupID	**GroupId**	**jms.Gid**	See Notes 4 and 5.
JMSXGroupSeq	**MsgSeqNumber**	**jms.Seq**	
JMSXUserID	**UserIdentifier**		

Note 4: The MQMD **GroupId** field can hold a standard WebSphere MQ **GroupId** of 48 hexadecimal digits (24 bytes). The **JMSXGroupID** is a string value containing hexadecimal characters and prefixed with "ID:" or an arbitrary string value not beginning "ID:". The first of these represents a standard WebSphere MQ **GroupId** and maps directly to or from the MQMD **GroupId** field (truncated or padded with zeros as applicable). The second (arbitrary string) uses the MQRFH2 **jms.Gid** field; the first 24 bytes of the string, in UTF-8 format, are written into the MQMD **GroupId**.

Note 5: When service integration is converting messages to WebSphere MQ format, if the **JMSXGroupID** has been set then service integration also sets the **MQMF_MSG_IN_GROUP** flag in the **MsgFlags** field of the MQMD. Note that when sending group messages, the sending JMS application must ensure that the **MQMF_LAST_MSG_IN_GROUP** flag is set as required (see "WebSphere MQ message properties: JMS provider-specific properties").

WebSphere MQ message properties: JMS provider-specific properties

The following table shows how service integration maps JMS provider-specific properties to and from MQMD and MQRFH2 fields when converting messages to and from WebSphere MQ format. Typically you use these properties to satisfy special requirements in the receiving application, so you should consult the developer or administrator of the receiving application for details of the required property values.

Table 80. JSM provider-specific properties with MQMD and MQRFH2 fields. The first column of this table lists the JMS provider-specific properties, and the second column shows the MQMD fields that relate to the JMS provider-specific properties in the first column. The third column shows the MQRFH2 fields that relate to the JMS provider-specific properties in the first column. The fourth column provides links, where required, to the footnotes that appear after the table.

JMS provider-specific property	MQMD field	MQRFH2 field	Notes
JMS_IBM_ArmCorrelator		**mqext.Arm**	See Note 6.
JMS_IBM_Character_Set	**CodedCharacterSetId**	**CodedCharacterSetId**	See Note 7.
JMS_IBM_Encoding	**Encoding**	**Encoding**	See Note 7.
JMS_IBM_Feedback	**Feedback**		
JMS_IBM_Format	**Format**	**Format**	See Note 7.
JMS_IBM_Last_Msg_In_Group	**MQMF_LAST_MSG_IN_GROUP**		See Note 8.
JMS_IBM_MQMD_CorrelId	**CorrelId**		See Notes 9 and 10.
JMS_IBM_MQMD_MsgId	**MsgId**		See Notes 9 and 11.
JMS_IBM_MQMD_Persistence	**Persistence**		See Notes 9, 12, and 15.
JMS_IBM_MQMD_ReplyToQ	**ReplyToQ**		See Notes 9, 13, and 16.
JMS_IBM_MQMD_ReplyToQMgr	**ReplyToQMgr**		See Notes 9, 13, and 16.
JMS_IBM_MsgType	**MsgType**		
JMS_IBM_PutDate	**PutDate**		

Table 80. JSM provider-specific properties with MQMD and MQRFH2 fields (continued). The first column of this table lists the JMS provider-specific properties, and the second column shows the MQMD fields that relate to the JMS provider-specific properties in the first column. The third column shows the MQRFH2 fields that relate to the JMS provider-specific properties in the first column. The fourth column provides links, where required, to the footnotes that appear after the table.

JMS provider-specific property	MQMD field	MQRFH2 field	Notes
JMS_IBM_PutTime	**PutTime**		
JMS_IBM_Report_*	**Report**		See Note 14.
JMS_IBM_RMCorrelator		**mqext.Wrm**	
JMS_TOG_ARM_Correlator		**mqext.Arm**	See Note 6.

Note 6: You should use the name **JMS_TOG_ARM_Correlator** for the ARM correlator. The name **JMS_IBM_ArmCorrelator** is available for compatibility with some existing JMS programs.

Note 7: The **JMS_IBM_Character_Set**, **JMS_IBM_Encoding**, and **JMS_IBM_Format** properties contain information about the WebSphere MQ message payload; that is, the part of the WebSphere MQ message that follows the MQRFH2 (if there is one) or the whole WebSphere MQ message, excluding the MQMD, if there is no MQRFH2. For more information about these properties and how to use them, see “Mapping the message body to and from WebSphere MQ format” on page 594.

Note 8: **MQMF_LAST_MSG_IN_GROUP** is one of the flags in the **MsgFlags** field of the MQMD.

Note 9: The **JMS_IBM_MQMD_CorrelId**, **JMS_IBM_MQMD_MsgId**, **JMS_IBM_MQMD_Persistence**, **JMS_IBM_MQMD_ReplyToQ**, and **JMS_IBM_MQMD_ReplyToQMgr** properties allow JMS applications to override the service integration default processing of WebSphere MQ MQMD fields. When service integration converts messages to WebSphere MQ format, service integration sets the corresponding MQMD field for each of these properties if, and only if, that property has been explicitly set by the application (using `setObjectProperty()` or `setNonNullProperty()`).

Service integration sets each of these properties from the corresponding MQMD field when it converts a message from WebSphere MQ format.

Note 10: The **JMS_IBM_MQMD_CorrelId** property overrides the default processing of the **JMSCorrelationID** property. When service integration converts messages to WebSphere MQ format, service integration sets the MQMD **CorrelId** field to the value (`byte[]`) if explicitly set of the **JMS_IBM_MQMD_CorrelId** property, regardless of the value (if any) of the **JMSCorrelationID** property. Setting the **JMS_IBM_MQMD_CorrelId** property does not affect the value of the MQRFH2 **jms.Cid** field.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_CorrelId** property to the value (`byte[]`) of the **MQMD CorrelId** field, regardless of the value (if any) of the MQRFH2 **jms.Cid** field.

Note 11: The **JMS_IBM_MQMD_MsgId** property overrides the JMS default processing of the **JMSMessageID** property. When service integration converts messages to WebSphere MQ format, service integration checks whether the **JMS_IBM_MQMD_MsgId** property has been explicitly set. If so, service integration sets the MQMD **MsgId** field to this value (byte[]) , and replaces the unique value of the **JMSMessageID** that JMS allocates to the message.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_MsgId** property to the value (`byte[]`) of the MQMD **MsgId** field.

Note 12: The **JMS_IBM_MQMD_Persistence** property overrides the default processing of the **JMSDeliveryMode** property. When service integration converts messages to WebSphere MQ format, service integration sets the MQMD **Persistence** field to the value (integer) if explicitly set of the **JMS_IBM_MQMD_Persistence** property, regardless of the value (if any) of the **JMSDeliveryMode** property. Setting the **JMS_IBM_MQMD_Persistence** property does not affect the value of the MQRFH2 **jms.Dlv** field.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_Persistence** property to the value (integer) of the MQMD **Persistence** field, regardless of the value (if any) of the MQRFH2 **jms.Dlv** field.

Note 13: The **JMS_IBM_MQMD_ReplyToQ** and **JMS_IBM_MQMD_ReplyToQMgr** properties override the default processing of the **JMSReplyTo** property. When service integration converts messages to WebSphere MQ format, service integration sets the MQMD **ReplyToQ** field to the value (string) if explicitly set of the **JMS_IBM_MQMD_ReplyToQ** property and the MQMD **ReplyToQMgr** field to the value (string) if explicitly set of the **JMS_IBM_MQMD_ReplyToQMgr** property, regardless of the value (if any) of the **JMSReplyTo** property. Setting the **JMS_IBM_MQMD_ReplyToQ** or **JMS_IBM_MQMD_ReplyToQMgr** field does not affect the value of the MQRFH2 **jms.Rto** field.

When service integration converts messages from WebSphere MQ format, service integration sets the **JMS_IBM_MQMD_ReplyToQ** and **JMS_IBM_MQMD_ReplyToQMgr** property to the values (string) of the MQMD **ReplyToQ** and **ReplyToQMgr** fields, regardless of the value (if any) of the MQRFH2 **jms.Rto** field.

Note 14: For a list of the **JMS_IBM_Report_*** properties, see “Mapping MQMD Report fields to JMS provider-specific properties” on page 605.

Note 15: For more information see “Mapping the JMS delivery option and message reliability to and from the WebSphere MQ persistence value.”

Note 16: For more information see “Mapping destinations to and from WebSphere MQ queues, topics, and destinations” on page 601.

Mapping the JMS delivery option and message reliability to and from the WebSphere MQ persistence value

When converting messages between WebSphere MQ format and service integration format, service integration processes the message header fields and properties relating to message delivery mode, reliability, and persistence.

Quality of service indications in the WebSphere MQ format message

The WebSphere MQ format message contains the following indications of the quality of service:

MQMD persistence

This property is present in all WebSphere MQ format messages. The property specifies the quality of service that WebSphere MQ provides for the message as follows:

MQMD persistence	Quality of service
MQPER_PERSISTENT	Persistent quality of service. WebSphere MQ assures once and once only delivery of the message.
MQPER_NOT_PERSISTENT	Nonpersistent quality of service. WebSphere MQ can discard the message in exceptional circumstances.
MQPER_PERSISTENCE_AS_Q_DEF	WebSphere MQ sets the quality of service (persistent or nonpersistent) to the value configured for the destination queue.

For more information about the WebSphere MQ persistent and nonpersistent qualities of service, see the WebSphere MQ documentation .

MQRFH2 jms.Dlv (JMSDeliveryMode)

This property is present in WebSphere MQ format JMS messages that include the MQRFH2 header. The property contains the **JMSDeliveryMode** that was set when the application issued `send` for the message.

Quality of service indications when service integration converts a message to WebSphere MQ format

Message reliability levels - JMS delivery mode and service integration quality of service describes how service integration sets the message reliability. Briefly: JMS applications send messages with a JMS delivery mode (`persistent` or `nonpersistent`), then service integration uses JMS connection factory settings to map the JMS delivery mode to a service integration message reliability setting, and finally additional settings on bus destinations (including foreign destinations and alias destinations) can override this message reliability. When service integration converts the message to WebSphere MQ format, it sets the **MQMD persistence** indicator as follows:

Service integration message reliability	MQMD persistence
Reliable persistent	Persistent
Assured persistent	Persistent
Reliable nonpersistent	Nonpersistent
Express nonpersistent	Nonpersistent
Best effort nonpersistent	Nonpersistent

The sending application can optionally override this by setting the **JMS_IBM_MQMD_Persistence** message property in the message.

Quality of service indications when service integration converts a message from WebSphere MQ format

When service integration receives a message from WebSphere MQ, it uses the **MQMD persistence** value of the message together with the corresponding service integration reliability value that you configure in the WebSphere MQ link receiver or the WebSphere MQ server queue point to determine the reliability of the service integration message.

Mapping destinations to and from WebSphere MQ queues, topics, and destinations

Service integration messages and WebSphere MQ messages both contain header fields and properties. Some of these header fields and properties contain destinations or destination properties that provide information about send-to and reply-to destinations, and about destinations in the bus forward and reverse routing paths. Because service integration and WebSphere MQ have different definitions for destinations, mappings are used to process the destinations and destination properties when messages are converted between service integration format and WebSphere MQ format.

Destinations and destination properties in the WebSphere MQ format message

The WebSphere MQ format message contains the following information related to destinations:

MQXQH RemoteQName
MQXQH RemoteQMgrName

These fields are present in the MQXQH (the WebSphere MQ Transmission-queue header). The MQXQH is attached to messages only while they travel between WebSphere MQ queue managers, or

between WebSphere MQ queue managers and service integration buses across a WebSphere MQ link. Sending and receiving applications cannot access these fields.

The **RemoteQName** field contains the name of the send-to queue (WebSphere MQ) or send-to destination (service integration). The **RemoteQMgrName** field identifies the queue manager or queue-sharing group (WebSphere MQ) or service integration bus (service integration) where the send-to queue or destination is located. Usually, the **RemoteQMgrName** field contains the name of the remote queue manager, queue-sharing group, or service integration bus, but it can contain a WebSphere MQ queue manager alias or a service integration virtual queue manager name. Note that these fields are not used for topics.

The **RemoteQName** and **RemoteQMgrName** values can be up to a maximum length of 48 characters, and must conform to WebSphere MQ naming restrictions.

MQRFH2 jms.Dst (JMSDestination)

This field is present in WebSphere MQ format JMS messages that include the MQRFH2 header.

The **jms.Dst** field contains a serialized representation (a WebSphere MQ URI) of the send-to JMS destination that was set when the application issued send for the message. Refer to the WebSphere MQ library for more information about the WebSphere MQ URI format for JMS destinations.

When service integration sends a message that has a forward-routing path to WebSphere MQ, it adds an **ibmRoutingPath** attribute to this URI. The **ibmRoutingPath** value identifies the forward routing path from the service integration message. WebSphere MQ does not use the forward routing path, but the send-to destination might be in a remote service integration bus that can use the forward routing path.

MQMD ReplyToQ
MQMD ReplyToQMgr

These fields are present in all WebSphere MQ format messages.

If the sending application specifies a reply-to queue, the **ReplyToQ** field contains the name of the reply-to queue (WebSphere MQ) or reply-to destination (service integration), and the **ReplyToQMgr** field identifies the queue manager or queue-sharing group (WebSphere MQ) or service integration bus (service integration) where that queue is located. Usually, the **ReplyToQMgr** field contains the name of the queue manager, queue-sharing group, or service integration bus, but it can contain a WebSphere MQ queue manager alias or a service integration virtual queue manager name. If the sending application specifies a reply-to topic, or if it does not specify a reply-to destination, these fields contain null values.

The **ReplyToQ** and **ReplyToQMgr** values can be up to a maximum length of 48 characters, and must conform to WebSphere MQ naming restrictions.

MQRFH2 jms.Rto (JMSReplyTo)

This field is present in WebSphere MQ format JMS messages that include the MQRFH2 header and that specify a reply-to destination. WebSphere MQ JMS applications usually use this message attribute as the destination for reply messages, but other (non-JMS) WebSphere MQ applications usually do not use it; they use the MQMD reply-to fields instead.

The **jms.Rto** field contains a serialized representation (a WebSphere MQ URI) of the reply-to JMS destination that is set by the sending JMS application. Refer to the WebSphere MQ library for more information about the WebSphere MQ URI format for JMS destinations.

When service integration sends a message that has a reverse-routing path to WebSphere MQ, service integration adds an **ibmRoutingPath** attribute to this URI. The **ibmRoutingPath** value identifies the reverse-routing path from the service integration message. When the receiving JMS application sends a reply, WebSphere MQ includes the routing path information from the reply-to URI in the send-to URI of the reply message so that service integration can use it for routing the reply message.

Note:

- When the sending application specifies a reply-to queue for a message, that queue is usually located in the bus, queue manager, or queue-sharing group to which the sending application connects. This allows the sending application to receive the reply message from the reply-to queue. Service integration applications that send messages to or through WebSphere MQ should not specify a reply-to queue in a different bus, queue manager, or queue-sharing group.
- It is important to understand that it is the receiving application which uses the reply-to destination in a message. The bus, queue manager, or queue-sharing group to which the receiving application connects must be configured with the information that allows routing to the reply-to destination.

Destination conversion when service integration converts a message to WebSphere MQ format

When service integration converts a message to WebSphere MQ format, it puts the following destination information into the WebSphere MQ format message:

MQXQH RemoteQName
MQXQH RemoteQMgrName

These fields apply only when service integration is sending the message across a WebSphere MQ link, and only when the destination is a queue.

Service integration sets these fields based on the resolved send-to destination for the message; that is, if the send-to destination is an alias, service integration uses the target bus and target identifier. Processing is then as follows:

- If the send-to destination is in an indirectly-connected bus, it stores the destination name (identifier) as the **RemoteQName** and the bus name as the **RemoteQMgrName**.
- If the send-to destination is in the directly-connected bus and the destination name (identifier) is in the form *queue@queueManager*, it stores the queue name (*queue*) as the **RemoteQName** and the queue manager name (*queueManager*) as the **RemoteQMgrName**.
- If the send-to destination is in the directly-connected bus and the destination name (identifier) is not in the form *queue@queueManager*, it stores the destination name as the **RemoteQName** and the bus name as the **RemoteQMgrName**.

If the send-to destination is in an indirectly-attached service integration bus and its destination name does not comply with WebSphere MQ naming restrictions, you must define an alias destination with a compliant name; the sending application must use the compliant (alias) name. In this case, you must define the alias destination in the remote (indirectly-attached) bus, not the local bus.

If the send-to destination is in an indirectly-attached service integration bus and its bus name does not comply with WebSphere MQ naming restrictions, there must be a virtual queue manager name for the indirectly-attached bus. In this case, the local bus must define the indirectly-attached bus with its virtual queue manager name, not its bus name.

For more information about mapping service integration bus names that do not comply with WebSphere MQ naming restrictions, see How to address bus destinations and WebSphere MQ queues.

MQRFH2 jms.Dst (JMSDestination)

If the WebSphere MQ format message includes the MQRFH2 header, service integration serializes the **JMSDestination** header field into a WebSphere MQ URI and stores it in the **JMSDestination** field in the WebSphere MQ message. If the message has a forward-routing path, service integration includes that in the URI as the **ibmRoutingPath** attribute.

MQMD ReplyToQ
MQMD ReplyToQMgr

The sending JMS application can set these fields directly by using the provider-specific JMS message properties **JMS_IBM_MQMD_ReplyToQ** and **JMS_IBM_MQMD_ReplyToQMgr**. If the sending application does not do this, service integration sets the properties if (and only if) the message has a reply-to destination and that destination is a queue.

Service integration sets these fields based on the unresolved reply-to destination for the message; that is, if the reply-to destination is an alias, service integration uses the alias bus and identifier, not the target bus and identifier. Service integration applications should not provide reply-to destinations that are foreign destinations or have names that include the @ character. Provided applications do not do this, processing is as follows:

- The reply-to destination name (identifier) is stored in the **ReplyToQ** field and the reply-to destination bus name (that is, the local bus name) in the **ReplyToQMgr** field.
- If the virtual queue manager name is different from the local bus name, the virtual queue manager name is stored in the **ReplyToQMgr** field instead of the local bus name.

If the real name of the reply-to destination does not comply with WebSphere MQ naming restrictions (including if the name includes the @ character), you must define an alias destination with a compliant name and the sending application must use the compliant (alias) name. In this case, you must define the alias destination in the local bus, not in the remote (indirectly-attached) bus.

MQRFH2 jms.Rto (JMSReplyTo)

If the message has a reply-to destination and the WebSphere MQ format message includes the MQRFH2 header, service integration constructs a WebSphere MQ URI to represent that reply-to destination, and stores the URI in the **JMSReplyTo** property in the WebSphere MQ message. If the reply-to destination is a queue, the URI includes the reply-to destination bus name (the local bus) or the virtual queue manager name (if that is different). If the message has a reverse-routing path, service integration includes that path in the URI in the **ibmRoutingPath** attribute.

Destination conversion when service integration converts a message from WebSphere MQ format

When service integration converts a message from WebSphere MQ format, it uses the following destination information from the WebSphere MQ format message:

MQXQH RemoteQName
MQXQH RemoteQMgrName

These fields are applicable only when service integration is receiving the message across a WebSphere MQ link, and only when the destination is a queue.

Service integration interprets the **RemoteQName** field as the destination identifier (always a queue) for the message, and the **RemoteQMgrName** field as the name of the destination bus for the message. If the **RemoteQMgrName** field contains the virtual queue manager name of the local bus, service integration interprets it as the name of the local bus. Service integration then uses the resulting bus and destination identifier combination (which can be an alias destination in the local bus) to deliver the message in the usual way. For the case where the destination bus is a foreign bus, this includes forwarding the message to that foreign bus.

MQRFH2 jms.Dst (JMSDestination)

If this field is available, service integration uses it to create the **JMSDestination** header field for the message. If the URI includes the **ibmRoutingPath** attribute, service integration uses that to create the forward-routing path for the message.

If this property is not available (for example, if the WebSphere MQ message has no MQRFH2 header), service integration can create a **JMSDestination** header field from the service integration destination where the message is delivered.

MQMD ReplyToQ
MQMD ReplyToQMgr

If these fields contain non-null values, service integration uses them to construct the first element of the reverse-routing path for the service integration message, as follows:

- If **ReplyToQMgr** is the local bus name or virtual queue manager name, service integration sets the bus to the local bus and the destination name (identifier) to **ReplyToQ**.
- If **ReplyToQMgr** is a foreign bus defined in the local bus, service integration sets the bus to **ReplyToQMgr** and the destination name (identifier) to **ReplyToQ**.
- If **ReplyToQMgr** is not the local bus name, virtual queue manager name, or a foreign bus defined in the local bus, service integration sets the bus to the directly-attached WebSphere MQ bus and the destination name (identifier) to *queue@queueManager* where *queue* is **ReplyToQ** and *queueManager* is **ReplyToQMgr**.

MQRFH2 jms.Rto (JMSReplyTo)

If this field is available, service integration uses it with the MQMD **ReplyToQ** and **ReplyToQMgr** fields to construct the reverse-routing path and **JMSReplyTo** header field for the service integration message. It constructs the reverse-routing path from the first element (which it constructs from the MQMD **ReplyToQ** and **ReplyToQMgr** fields) and any remaining elements that it obtains from the **ibmRoutingPath** attribute (if there is one) of the **JMSReplyTo** URI in the WebSphere MQ message. It constructs the **JMSReplyTo** header field from the first element of the reverse-routing path together with the destination attributes of the **JMSReplyTo** URI in the WebSphere MQ message.

Mapping MQMD Report fields to JMS provider-specific properties

JMS applications can use report messages as a form of managed request/response processing, to give remote feedback to producers on the outcome of their send operations and the fate of their messages. A JMS application can request different types of report message by setting JMS_IBM_Report_Xxxx message properties and options.

JMS applications can request the following types of report message by setting the appropriate JMS_IBM_Report_Xxxx message properties and options. The options have the same general syntax and meaning:

MQRO_*report-type*
A report message of the indicated type is generated that contains the WebSphere MQ message descriptor (MQMD) of the original message. It does not contain any message body data.

MQRO_*report-type*_**WITH_DATA**
A report message of the indicated type is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_*report-type*_**WITH_FULL_DATA**
A report message of the indicated type is generated that contains all data from the original message.

Use the following prefix with each option: com.ibm.websphere.sib.api.jms.

For example, to request a Confirm on delivery (COD) report message with full data, the JMS application must set JMS_IBM_Report_COD to the value com.ibm.websphere.sib.api.jms.MQRO_COD_WITH_FULL_DATA.

For each type of report message, the following table shows the JMS_IBM_Report_Xxxx message property that a JMS application can set, and the MQMD Report field options that map to the property.

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Exception	Send a report message if the request message cannot be put to the target queue. The exception report messages are generated when a message has been rerouted to an exception destination.	JMS_IBM_Report_Exception • MQRO_EXCEPTION • MQRO_EXCEPTION_WITH_DATA • MQRO_EXCEPTION_WITH_FULL_DATA
Discard	Discard the original request message rather than sending it to an exception destination. You can use this option with the JMS_IBM_Report_Exception property set to MQRO_EXCEPTION_WITH_FULL_DATA to return an undeliverable request message to its sender.	JMS_IBM_Report_Discard_Msg • TRUE • FALSE
Expiration	Send a report message if the request message passes its expiry time.	JMS_IBM_Report_Expiration • MQRO_EXPIRATION • MQRO_EXPIRATION_WITH_DATA • MQRO_EXPIRATION_WITH_FULL_DATA
Confirm on arrival (COA)	Send a report message when the request message has been put to the target queue. For publish/subscribe messaging, the COA report message is generated only on the producers messaging engine. Therefore, such reports are relevant only to local subscriptions. For point-to-point messaging, COA messages are generated when the message arrives at the final destination. For partitioned queues, the report message is generated only when the put operation has committed and a final destination has therefore been selected. Any With_Data or With_Full_Data report options specified are ignored; the COA report message deals only with message headers. If a forward-routing path is used, the COA message are generated when the message arrives at the final destination in the path.	JMS_IBM_Report_COA • MQRO_COA • MQRO_COA_WITH_DATA • MQRO_COA_WITH_FULL_DATA

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Confirm on delivery (COD)	Send a report message when the request message has been removed from the queue or topic space by a message consumer. For publish/subscribe messaging, the COD message is generated when all subscribers have received the request message. Therefore, there is one COD message generated for every COA. When a message is consumed by a subscriber, the reference count of the message on the topic space is reduced. When the reference count reaches zero, the message is removed from the topic space then a COD report message is generated. For point-to-point messaging, the COD message is generated after the message has been successfully received by a consuming application. Any With_Data or With_Full_Data report options specified are ignored; the COD report message deals only with message headers.	JMS_IBM_Report_COD • MQRO_COD • MQRO_COD_WITH_DATA • MQRO_COD_WITH_FULL_DATA
Positive action notification (PAN)	Ask the consumer application to send a report message when it has successfully processed the request message.	JMS_IBM_Report_PAN • TRUE • FALSE
Negative action notification (NAN)	Ask the consumer application to send a report message if it has not successfully processed the request message.	JMS_IBM_Report_NAN • TRUE • FALSE

The requesting application can control other aspects of the report message as follows:

- How the message Id is generated for the report message and any reply message:

  MQRO_New_Msg_Id
  This the default. A new message Id is generated for the report message.

  MQRO_Pass_Msg_Id
  The message Id of the report message is set to the message Id of the request message.

- How the correlation Id of the report or reply message is to be set.

  MQRO_Copy_Msg_Id_To_Correl_Id
  This the default. the correlation Id of the report message is set to the message Id of the request message.

  MQRO_Pass_Correl_Id
  The correlation Id of the report message is set to the correlation Id of the request message.

For more information about report messages and the associated properties and options refer to the *Using Java* section of the WebSphere MQ information center, available from the WebSphere MQ library.

Mapping additional MQRFH2 header fields in service integration

In the WebSphere MQ message header (MQRFH2), there are additional fields that are specific to the service integration bus and that allow for functions that are not used in WebSphere MQ. When

WebSphere MQ transports a message from one service integration bus to another service integration bus, these fields convey information that can be used by service integration applications but is not required by WebSphere MQ.

The additional fields are inserted in the MQRFH2 header of application messages, in the <sib> and <jms> folders. These fields do not appear as JMS message fields or properties.

When a message is sent to WebSphere MQ, a <sib> folder is included in the MQRFH2 header of the message if both of the following are true:

- The WebSphere MQ queue point attributes of the service integration destination are configured to use MQRFH2 headers.
- The fields that correspond to the <sib> folder content are set in the service integration message.

MQRFH2 header and field (<jms> folder)	SIBusMessage field or property
Frp (appended to Dst field)	Forward routing path header field
Rrp (appended to Rto field)	Reverse routing path header field

MQRFH2 header and field (<sib> folder)	SIBusMessage field or property
RTopic	Reply topic
RPri	Reply priority
RPer	Reply persistence
RTTL	Reply time to live
JsApiUserId	Application user ID (JMSXUserId) for the service integration application
JsDst	JMS destination
JsFmt	Message format
JsSysMsgId	System message identifier

Mapping the JMS Destination property between service integration and WebSphere MQ

The properties of service integration destinations differ from those used by WebSphere MQ queues, and they cannot be mapped exactly. When service integration uses WebSphere MQ to transport a message, it is useful to keep both representations of the JMS destination property in the message. To address this, when a message leaves service integration and enters WebSphere MQ, an additional **RFH2** property is introduced into the RFH2 header to store the service integration destination property.

The service integration destination property is serialized, formatted as a hexadecimal string, then stored using the **JsDst** property of the service integration RFH2 folder, the <sib> folder. WebSphere MQ applications do not make use of this folder, but if the message is to be retrieved by another service integration application, it can use the information.

For example, a service integration destination SIQ1 is localized on a WebSphere MQ queue MQQ1, residing on queue manager QM1. The following actions occur when a service integration application sends a message to SIQ1:

- A serialized representation of PMQ1 is placed in the <sib> folder of the RFH2 header, using the **JsDst** property.
- The message is stored on MQQ1.
- The string "queue://QM1/MQQ1" is also placed in the <jms> folder of the RFH2 header, using the **Dst** property.

This follows the convention used by the WebSphere MQ messaging provider to encode JMS destinations. If the message is retrieved by a service integration application, the JMS destination can be recovered from the <sib> folder of the RFH2 header. If the message is retrieved by a WebSphere MQ application, the JMS destination can be recovered from the contents of the <jms> folder of the RFH2 header.

Note: If a WebSphere MQ server bus member is configured so that it does not use RFH2 headers, the JMS destination is not preserved when the message enters WebSphere MQ. In this situation, a service integration application can still retrieve the JMS message, but any attempt to examine the JMS destination property causes a JMS exception.

How to process WebSphere MQ message headers

WebSphere MQ messages can optionally include additional headers, or alternative headers, to the MQRFH2 header, which contains JMS properties. WebSphere Application Server application programs can use the com.ibm.mq.headers classes to access headers in messages from WebSphere MQ and to construct headers in messages to WebSphere MQ.

WebSphere MQ message headers

WebSphere MQ messages always include a message descriptor (MQMD). They can also include headers that contain additional information about the messages; for example, messages to or from JMS applications usually include an MQRFH2 header that contains message properties. WebSphere MQ defines the format and usage of some headers (for example, MQRFH2) and also allows users and third-party software providers to define their own custom headers.

Typically it is not necessary for application programs to process WebSphere MQ message headers. Most WebSphere MQ applications either do not use headers at all, or only use the MQRFH2 header, and the service integration and WebSphere MQ messaging providers automatically process the MQRFH2 header when you are communicating with these applications. However, if you are communicating with a WebSphere MQ application that uses or creates additional or different headers, then your WebSphere Application Server application can use the com.ibm.mq.headers classes to create the headers in messages it sends, and process them in messages it receives.

In the WebSphere MQ message, the headers (if there are headers) are at the start of the message, before the message payload. Each header contains fields that describe the following header, or the payload if there are no more headers; the MQMD contains the fields that describe the first header, or the payload if there are no headers. The MQMD and the MQRFH2 header do not normally appear in a JMS message. When the messaging provider converts a WebSphere MQ message into a JMS message, it uses information from the MQMD and MQRFH2 header to set JMS header fields and properties. Similarly, when the messaging provider converts a JMS message into a WebSphere MQ message, it uses the JMS header fields and properties to construct the MQMD and MQRFH2 header.

The JMS provider handles other headers in WebSphere MQ messages by converting the WebSphere MQ message to or from a JMS BytesMessage; the headers appear at the start of the message body, followed by the WebSphere MQ message payload (if any). The **JMS_IBM_Format** property of the JMS message indicates the format of the data in the message body (in this case, first header) and the **JMS_IBM_Encoding** and **JMS_IBM_Character_Set** properties indicate its encoding and CCSID.

Processing WebSphere MQ message headers in a JMS BytesMessage

The com.ibm.mq.headers package contains classes and interfaces that you can use to parse and manipulate WebSphere MQ headers in the body of a JMS BytesMessage. The MQHeader interface provides general-purpose methods for accessing header fields and for reading and writing message content. Each header type has its own class that implements the MQHeader interface and adds getter and setter methods for individual fields. For example, the MQCIH class represents the MQCIH (CICS Bridge) header type. The header classes perform any necessary data conversion automatically, and can read or write data in any specified numeric encoding or character set (CCSID).

Two helper classes, MQHeaderIterator and MQHeaderList, assist with reading and decoding (parsing) the header content in messages:

- MQHeaderIterator works like a java.util.Iterator. For as long as there are more headers in the message, the `next()` method returns `true`, and the `nextHeader()` or `next()` method returns the next header object.
- MQHeaderList works like a java.util.List. Like MQHeaderIterator, it parses header content, but it also allows you to search for particular headers, add new headers, remove existing headers, update header fields, and then write the header content back to a message. Alternatively, you can create an empty MQHeaderList, then populate it with header instances and write it to a message once or repeatedly.

Every header class implements the MQHeader interface, which provides the methods `int read (java.io.DataInput message, int encoding, int characterSet)` and `int write (java.io.DataOutput message, int encoding, int characterSet)`. The java.io.DataInputStream and java.io.DataOutputStream classes implement DataInput and DataOutput respectively. You can obtain DataInput and DataOutput objects from byte arrays carried in JMS messages, as in the following example, which processes a single MQCIH header:

```
import java.io.*;
import javax.jms.*;
import com.ibm.mq.headers.*;
...
BytesMessage bytesMessage = (BytesMessage) msg; // Message received from JMS consumer
byte[] bytes = new byte [(int) bytesMessage.getBodyLength ()];
bytesMessage.readBytes (bytes);
DataInput in = new DataInputStream (new ByteArrayInputStream (bytes));
MQCIH cih = new MQCIH (in,
   bytesMessage.getIntProperty("JMS_IBM_Encoding"),
   819);
```

Alternatively, you can use the MQHeaderIterator class to process a sequence of headers, replacing the line starting `MQCIH cih = new MQCIH` with:

```
MQHeaderIterator it = new MQHeaderIterator (in,
   bytesMessage.getStringProperty("JMS_IBM_Format"),
   bytesMessage.getIntProperty("JMS_IBM_Encoding"),
   819);
while (it.hasNext()) {
   MQHeader item = (MQHeader) it.next();
   ...
}
```

This example creates a single header (an MQCIH type header) and adds it into a BytesMessage:

```
import java.io.*;
import javax.jms.*;
import com.ibm.mq.constants.CMQC;
import com.ibm.mq.headers.*;
...
MQCIH header = new MQCIH();
ByteArrayOutputStream out = new ByteArrayOutputStream ();

header.write (new DataOutputStream (out), CMQC.MQENC_NATIVE, 819);
byte[] bytes = out.toByteArray ();
BytesMessage newMsg = origSes.createBytesMessage();
newMsg.writeBytes(bytes);
```

This example uses the MQHeaderList class to add two headers into a BytesMessage:

```
import java.io.*;
import javax.jms.*;
import com.ibm.mq.constants.CMQC;
import com.ibm.mq.headers.*;
...
byte[] outheaders = null;
byte[] outbody = ...
try {
```

```
    MQHeaderList it = new MQHeaderList ();
    MQHeader header1 = ... // Could be any header type
    MQHeader header2 = ... // Could be any header type
    ByteArrayOutputStream out = new ByteArrayOutputStream ();
    DataOutput dout = new DataOutputStream(out);
    it.add(header1);
    it.add(header2);
    it.write(dout);
    outheaders = out.toByteArray();
} catch (Exception e) {
    System.out.println("error generating MQ message headers : " + e);
}
BytesMessage newMsg = origSes.createBytesMessage();
newMsg.writeBytes(outheaders);
newMsg.writeBytes(bytes);

newMsg.setStringProperty("JMS_IBM_Format", "MQCICS");
newMsg.setIntProperty("JMS_IBM_Encoding", CMQC.MQENC_NATIVE);
newMsg.setIntProperty("JMS_IBM_Character_Set", 819);
```

Always use the correct values for the encoding and characterSet arguments. When you read headers, specify the encoding and CCSID with which the byte content was originally written. When writing headers, specify the encoding and CCSID that you want to produce. The data conversion is performed automatically by the header classes.

More information about the com.ibm.mq.headers classes

The com.ibm.mq.headers package is part of the WebSphere MQ Resource Adapter, which is installed automatically in WebSphere Application Server. This package comprises a set of classes and interfaces that allow the Java programmer to work with WebSphere MQ Message Headers. These include the MQHeaderIterator and MQHeaderList classes mentioned in this topic, and classes for some commonly used WebSphere MQ Message Headers, including:

- MQCIH – CICS bridge header
- MQIIH – IMS information header
- MQSAPH – SAP header

You can also define classes representing your own headers.

WebSphere MQ functions not supported by service integration

There are various functions available in a WebSphere MQ network that are not available on a service integration bus.

The following list helps you identify those functions but it is given as guidance rather than a complete definition. Functions not supported include:

1. Native MQ client (this includes client applications that make use of the base MQ classes for Java) attach.
2. Message segmentation.
3. Message grouping.
4. The MQMD Offset. Original length, MsgFlags, MsgSeqNumber, and GroupId fields are not supported because Message grouping and message segmentation are not supported.
5. Distribution lists.
6. Reference messages.
7. Triggering.
8. Alternate user authority.
9. Pass/set identity context.
10. In a program, setting the attributes of a queue (that is, the equivalent function of MQSET).

11. Confirmation of arrival/delivery.
12. Cluster sender/receiver channels (and cluster workload exits), because a messaging engine cannot participate in a WebSphere MQ cluster.
13. Server and requestor channels.
14. API crossing exits.
15. Data conversion exits.
16. Channel exits.
17. The equivalent to the MCAUSER and PUTAUTH fields of a channel.
18. Networks based on NetBIOS, SPX or SNA.
19. Message based command server.
20. PCF (Programmable Canonical Form messages).
21. Model queues. Service integration does not allow you to define model queues of a given name. Service integration technology supports only one model queue called the SYSTEM.DEFAULT.MODEL.QUEUE.
22. Dynamic queue name prefix length. Service integration suffixes all dynamic queue names with "_Q" and a unique id. This restricts the name specified in the dynamic queue name field of the Object Descriptor to up to 12 chars. If this name is greater than 12 characters, then it is truncated to 12 characters. In service integration, it is not possible to create a dynamic queue with the full name specified in the dynamic queue name field of the Object Descriptor.
23. Mark skip backout option.
24. Signal option on a get request.
25. Version 3 get message options structures.
26. All queue properties (the properties of a service integration destination do not map, one for one, to the properties of a WebSphere MQ local queue, for example).
27. Poisoned messages. Service integration bus local destination definitions have a maximum failed deliveries count (that is, the equivalent to the WebSphere MQ BackoutThreshold value) but there is no equivalent of the WebSphere MQ backout requeue queue name. In service integration technology, poisoned messages are instead backed out to an exception destination. Additionally, in service integration technology, when the number of times an application backs out a poisoned message is equal to the maximum failed deliveries count, the message is automatically backed out to an ExceptionDestination. If there is more than one message in the current unit of recovery, only the poisoned message is backed out to the ExceptionDestination. The remainder of the messages in the unit of recovery are backed out to the destination from which they were read.
28. A strict limitation of 48 bytes on the name of a queue. Service integration bus destination names can be greater than 48 bytes in length. If a destination name is to be returned to a WebSphere MQ JMS application, then it is important to use 48 byte destination lengths. Though, in some cases, it might be feasible to define an alias destination with a name length of up to 48 bytes to map to a local destination with a name of length greater than 48 bytes.

Programming to use message-driven beans

Applications can use message-driven beans as asynchronous message consumers. You deploy a message-driven bean as a message listener for a destination. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination.

About this task

You can use a tool such as Rational Application Developer to develop applications that use message-driven beans. You can use the WebSphere Application Server runtime tools, such as the administrative console, to deploy and administer applications that use message-driven beans.

For more information about implementing enterprise applications that use message-drive beans, see the following topics:

Procedure

- Develop message-driven beans.
- Design an enterprise application to use message-driven beans.
- Develop an enterprise application to use message-driven beans.
- Deploy an enterprise application to use message-driven beans with JCA 1.5-compliant resources.
- Deploy an enterprise application to use message-driven beans with listener ports.

Developing message-driven beans

You can develop a bean implementation class for a message-driven bean as introduced by the Enterprise JavaBeans specification. A message-driven bean (MDB) is a message consumer that implements business logic and runs on the server.

Before you begin

Determine the messaging model you want for your application regarding use of topics, queues, producers and consumers, publish or subscribe, and so on. You can refer to the message-driven bean component contract that is described in the Enterprise JavaBeans™ specification.

About this task

A message-driven bean (MDB) is a consumer of messages from a Java Message Service (JMS) provider. An MDB is invoked on arrival of a message at the destination or endpoint that the MDB services. MDB instances are anonymous, and therefore, all instances are equivalent when not actively servicing a client message. The container controls the life cycle of bean instances, which hold no state that is visible to a client.

The following example is a basic message-driven bean:

```
@MessageDriven(activationConfig={
                @ActivationConfigProperty(propertyName="destination",     propertyValue="myDestination"),
                @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
public class MsgBean implements javax.jms.MessageListener {

  public void onMessage(javax.jms.Message msg) {

      String receivedMsg = ((TextMessage) msg).getText();
      System.out.println("Received message: " + receivedMsg);

   }

}
```

As with other enterprise bean types, you can also declare metadata for message-driven beans in the deployment descriptor rather than using annotations, for example:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar id="EJBJar_1060639024453" version="3.0"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
      metadata-complete="false">
  <enterprise-beans>

    <message-driven>

      <ejb-name>MsgBean</ejb-name>
      <ejb-class>com.acme.ejb.MsgBean</ejb-class>
```

```
      <activation-config>
         <activation-config-property>
            <activation-config-property-name>destination</activation-config-property-name>
            <activation-config-property-value>myDestination</activation-config-property-value>
         </activation-config-property>
         <activation-config-property>
           <activation-config-property-name>destinationType</activation-config-property-name>
           <activation-config-property-value>javax.jms.Queue</activation-config-property-value>
        </activation-config-property>
      </activation-config>

    </message-driven>

  </enterprise-beans>
</ejb-jar>
```

Procedure

- Code the business logic of the message-driven bean, which must implement the appropriate message listener interface defined by the messaging type; for example, javax.jms.MessageListener. The business logic is invoked when the message listener method of the MDB is called to service a message; for example, `MessageListener.onMessage()`. If the MDB implements more than one interface, denote the message listener interface by coding the `messageListenerInterface` attribute of the `MessageDriven` annotation, or by coding the `<messaging-type>` element of the `message-driven` deployment descriptor element. You do not have to specify which is the message listener interface, as long as there is only one interface other than `java.io.Serializable`, `java.io.Externalizable`, or any of the `javax.ejb` package interfaces.
- You can optionally define message destination references on any type of enterprise bean. A message destination reference is a logical name by which an enterprise bean can refer to a message destination. The `Resource` annotation is used to inject a message destination reference, for example:

  ```
  @Resource (name="jms/Outlet", type=javax.jms.Queue) Queue salesOutlet;
  ```

  Alternatively, you can use the `<message-destination-ref>` element in the deployment descriptor to specify the message destination reference; for example:

  ```
  <message-destination-ref>
     <message-destination-ref-name>jms/Outlet</message-destination-ref-name>
     <message-destination-type>javax.jms.Queue</message-destination-type>
     <injection-target>
       <injection-target-class>com.acme.ejb.MsgBean</injection-target-class>
       <injection-target-name>salesOutlet</injection-target-name>
    </injection-target>
  </message-destination-ref>
  ```

  The `message-destination-ref` element is similar to the `resource-env-ref` element, but also has subelements, `message-destination-usage` with possible values `Produces`, `Consumes` or `ProducesConsumes`, and `message-destination-link`. You can use the `message-destination-link` element to tie two or more message-destination-ref references in the deployment descriptor together, which allows the deployer to bind the destination for several enterprise beans all at once, to the same destination. The message-destination-link value must match the `message-destination-name` value in the `message-destination` element; for example:

  ```
  <ejb-jar>

    <enterprise-beans>

      <session>
        <ejb-name>OutletBean</display-name>
        ...
        <message-destination-ref>
          <message-destination-ref-name>jms/target</message-destination-ref-name>
          <message-destination-type>javax.jms.Queue</message-destination-type>
          <message-destination-usage>Produces</message-destination-usage>
          <message-destination-link>destination</message-destination-link>
        </message-destination-ref>
        ...
      </session>
  ```

```
    <session>
      <ejb-name>InletBean</display-name>
      ...
      <message-destination-ref>
        <message-destination-ref-name>jms/source</message-destination-ref-name>
        <message-destination-type>javax.jms.Queue</message-destination-type>
        <message-destination-usage>Consumes</message-destination-usage>
        <message-destination-link>destination</message-destination-link>
      </message-destination-ref>
      ...
    </session>


    <message-driven>
      <ejb-name>InletBean</display-name>
      ...

      <ejb-name>MsgBean</ejb-name>
      <ejb-class>com.acme.MsgBean</ejb-class>
      <messaging-type>javax.jms.MessageListener</messaging-type>
      <message-destination-type>javax.jms.Queue</message-destination-type>
      <message-destination-link>destination</message-destination-link>

      ...
    </message-driven>

  </enterprise-beans>
      ...
  <assembly-descriptor>
    ...
    <message-destination>
      <message-destination-name>destination</message-destination-name>
    </message-destination>
    ...
  </assembly-descriptor>

</ejb-jar>
```

The `message-destination-link` element can refer to a destination that is defined in a different Java archive (JAR) file within the same application, as with an `ejb-link` element. For example, to link to the destination, `ProduceQueue`, defined in the `grocery.jar` file, enter the following line in the deployment descriptor:

```
<message-destination-link>grocery.jar#ProduceQueue</message-destination-link>
```

- As with any enterprise bean, you can package a message-driven bean in a JAR file, or in a web application archive (WAR) file.

Results

You developed a simple message-driven bean, along with some deployment and packaging options.

What to do next

Read related information about designing an enterprise application that uses message-driven beans.

Designing an enterprise application to use message-driven beans

To help you design your enterprise application, consider a generic enterprise application that uses one message-driven bean to retrieve messages from a JMS queue destination, and passes the messages on to another enterprise bean that implements the business logic.

About this task

To design an enterprise application to use message-driven beans, complete the following steps:

Procedure

1. Identify the message listener interface for the message type that the message-driven bean is to handle. The message-driven bean class must implement this message listener interface. For example, an EJB message-driven bean class used for JMS messaging must implement the javax.jms.MessageListener interface.
2. Identify the resources that the application is to use. This helps to identify the properties of resources that must be used within the application and configured as application deployment descriptors or within WebSphere Application Server.

Table 81. JMS resource types and examples of their properties. The left hand column of this table lists the JMS resource types, and the right hand column shows examples of the properties of each of the JMS resource types shown in the left hand column.

JMS resource type	Properties (for example)
JMS connection factory	**Name:** SamplePtoPQueueConnectionFactory **JNDI Name:** Sample/JMS/QCF
JMS destination	**Name:** Q1 **JNDI Name:** Sample/JMS/Q1
J2C activation specification properties	**Name:** MyMDBsActivationSpec **JNDI Name:** eis/MyMDBsActivationSpec **Destination JNDI Name:** MyQueue **Destination type:** javax.jms.Queue
Message-driven bean (deployment properties)	**Name:** JMSppSampleMDBBean **Transaction type:** Container **Message selector:** JMSType='car' **Acknowledge mode:** Dups OK Acknowledge **Destination type:** javax.jms.Queue **ActivationSpec JNDI name:** MyMDBsActivationSpec
Business logic bean	**Name:** MyLogicBean

   Ensure that you use consistent values where needed; for example, the JNDI name for the J2C activation specification must be the same in both the activation specification and the Message-driven bean deployment attributes.
3. Separate out the business logic. You should develop a message-driven bean to delegate the business processing of incoming messages to another enterprise bean. This provides clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client.
4. Decide whether to configure security. Messages arriving at a destination being processed by a listener have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component. For more information about EJB security, see EJB component security.
5. Understand how best effort nonpersistent messages are handled by the default messaging provider.

   If you have a non-transactional message-driven bean, the system either deletes the message when the message-driven bean starts, or when the message-driven bean completes. If the message-driven bean generates an exception, and therefore does not complete, the system takes one of the following actions:

   - If the system is configured to delete the message when the message-driven bean completes, then the message is despatched to a new instance of the message-driven bean, so the message has another opportunity to be processed.
   - If the system is configured to delete the message when the message-driven bean starts, then the message is lost.

   The message is deleted when the message-driven bean starts if the quality of service is set to `Best effort nonpersistent`. For all other qualities of service, the message is deleted when the message-driven bean completes.

Developing an enterprise application to use message-driven beans

Applications can use message-driven beans as asynchronous message consumers. You deploy a message-driven bean as a message listener for a destination. The message-driven bean is invoked by an activation specification or a JMS listener when a message arrives on the input destination that is being monitored.

About this task

You develop an enterprise application to use a message-driven bean as with any other enterprise bean, except that a message-driven bean does not have a home interface or a remote interface.

You should develop your message-driven bean to delegate the business processing of incoming messages to another enterprise bean, which provides clear separation of message handling and business processing. This separation also means that the business processing can be invoked either by the arrival of incoming messages or, for example, by a WebSphere J2EE client. Responses can be handled by another enterprise bean acting as a sender bean, or they can be handled in the message-driven bean.

EJB 2.0 message-driven beans support only Java Message Service (JMS) messaging. EJB 2.1 and EJB 3 message-driven beans can support other messaging types in addition to JMS. All message-driven beans must implement the MessageDrivenBean interface. For JMS messaging, a message-driven bean must also implement the message listener interface javax.jms.MessageListener. Other Java EE Connector Architecture (JCA)-compliant resource adapters might provide their own message listener interfaces that must be implemented.

You can use the New Enterprise Bean wizard of Rational Application Developer to create an enterprise bean with a bean type of `Message-driven bean`. The wizard creates appropriate methods for the type of bean.

By convention, the message-driven bean class is named *name*Bean, where *name* is the name you assign to the message-driven bean; for example:

```
public class MyJMSppMDBBean implements MessageDrivenBean, javax.jms.MessageListener
```

A message-driven bean can be registered with the EJB timer service for time-based event notifications if it also implements the javax.ejb.TimedObject interface, and invokes the timer callback method by the following call: `void ejbTimeout(Timer)`. At the scheduled time, the container then calls the message-driven bean ejbTimeout method.

The message-driven bean class must define and implement the following methods:

- onMessage(message), which must meet the following requirements:
  - The method must have a single argument of type javax.jms.Message.
  - The `throws` clause must not define any application exceptions.
  - If the message-driven bean is configured to use bean-managed transactions, it must call the javax.transaction.UserTransaction interface to scope the transactions. Because these calls occur inside the onMessage() method, the transaction scope does not include the initial message receipt. For more information, see the topic about message-driven beans transaction support.

  To handle the message within the onMessage() method (for example, to pass the message on to another enterprise bean), you use standard JMS. This is known as bean-managed messaging.

  If you are using a JCA-compliant resource adapter with a different message listener interface, another method besides the onMessage() method might be needed. For information about the message listener interface needed, see the documentation that was provided with your JCA-compliant resource adapter.
- ejbCreate()

  You must define and implement an ejbCreate method for each way in which you want a new instance of an enterprise bean to be created.
- ejbRemove()

This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean home interface from the javax.ejb.EJBHome interface. This method must contain any code that you want to execute before an enterprise bean instance is removed from the container (and the associated data is removed from the data source).

- ejbTimeout(Timer)

  This method is needed only to support notifications from the timer service, and contains the business logic that handles time events received.

Procedure

1. Create the Enterprise Application project.
2. Create the message-driven bean class. You can use the New Enterprise Bean wizard of Rational(r) Application Developer to create the enterprise bean with a bean type of `Message-driven bean`.

   For an example of how to create the message-driven bean class, see the Example section of this topic. For more information, see *Creating message-driven beans* in the Rational Application Developer information center. The result of this step is a message-driven bean that can be assembled into an EAR file for deployment.
3. Optional: Use the EJB deployment descriptor editor to review and, if needed, change the deployment attributes. You can use the EJB deployment descriptor editor to review deployment attributes that you specified on the EJB creation wizard (such as **`Transaction type`** and **`Message selector`**) and other default deployment attributes.

   If needed, you can override the values of these attributes later, after the enterprise application has been exported into an EAR file for deployment, as described in “Configuring deployment attributes for a message-driven bean against JCA 1.5-compliant resources” on page 2085 and “Configuring deployment attributes for a message-driven bean against a listener port” on page 2088.

   a. In the property pane, select the **Bean** tab.

   b. On the main panel, configure the **`Transaction type`** attribute.

      `Transaction type`
      This attribute determines whether the message-driven bean manages its own transactions, or whether the container manages transactions on behalf of the bean.
      `Bean` The message-driven bean manages its own transactions.
      `Container`
      The container manages transactions on behalf of the bean.

   c. Under **Activation Configuration**, review the following attributes:

      `acknowledgeMode`
      This attribute determines how the session acknowledges any messages it receives.
      `Auto Acknowledge`
      The session automatically acknowledges delivery of each message.
      `Dups OK Acknowledge`
      The session lazily acknowledges the delivery of messages. This setting is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumer applications that are tolerant of duplicate messages.

      As defined in the EJB specification, clients cannot use the Message.acknowledge() method to acknowledge messages. If a value of `CLIENT_ACKNOWLEDGE` is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and the Message.acknowledge() method is not used.

      Note:

      The acknowledgement is sent when the message is deleted.

      If you have a non-transactional message-driven bean, the system either deletes the message when the message-driven bean starts, or when the message-driven bean

completes. If the message-driven bean generates an exception, and therefore does not complete, the system takes one of the following actions:

- If the system is configured to delete the message when the message-driven bean completes, then the message is despatched to a new instance of the message-driven bean, so the message has another opportunity to be processed.
- If the system is configured to delete the message when the message-driven bean starts, then the message is lost.

The message is deleted when the message-driven bean starts if the quality of service is set to `Best effort nonpersistent`. For all other qualities of service, the message is deleted when the message-driven bean completes.

destinationType
This attribute determines whether the message-driven bean uses a queue or topic destination.
Queue The message-driven bean uses a queue destination.
Topic The message-driven bean uses a topic destination.

subscriptionDurability
This attribute determines whether a JMS topic subscription is durable or nondurable.
Durable
A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription messages until they are received by the subscription or until they expire.
Nondurable
Nondurable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A nondurable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created.

messageSelector
This attribute determines the JMS message selector that is used to select which messages the message-driven bean receives. For example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

d. Specify bindings deployment attributes.

Under **WebSphere Bindings**, select the **JCA Adapter** option then specify the bindings deployment attributes:
ActivationSpec JNDI name
This attribute specifies the JNDI name of the activation specification that is used to deploy this message-driven bean. This name must match the name of an activation specification that you define to WebSphere Application Server.
ActivationSpec Authorization Alias
This attribute specifies the name of an authentication alias used for authentication of connections to the JCA resource adapter. An authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.
Destination JNDI name
This attribute specifies the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI namespace.

e. Optional: Specify **Destination Link** to enable message linking.

Message linking allows the routing of messages to a specific message-driven bean in a deployment. Message linking allows message flow to be orchestrated between components in the same application.

For a message to be consumed and processed by a message-driven bean, the<message-destination-link> element must be defined in the deployment descriptor associated with the message-driven bean. The destination identified by the <message-destination-link> element corresponds to the logical destination.

When the <message-destination-ref> includes a <message-destination-link> element, messages are consumed at that destination.

In order to get the message-driven bean to consume messages sent to a destination, you can declare a <message-destination-link> element in the deployment descriptor, or alternatively set it in the activation specification.

4. Assemble and package the application for deployment.

Results

The result of this task is an EAR file, containing the message-driven bean, for the enterprise application that can be deployed in WebSphere Application Server.

Example

The following example shows how to create the message-driven bean class. The example code shows how to access the text and the JMS **MessageID**, from a JMS message of type TextMessage. In this example, first the onMessage() method of a message-driven bean is used to unpack the incoming text message and to extract the text and message identifier; then a private putMessage method (defined within the same message bean class) is used to put the message onto another queue:

```
public void onMessage(javax.jms.Message msg)
{
        String text      = null;
        String messageID = null;

        try
        {
                text = ((TextMessage)msg).getText();

                System.out.println("senderBean.onMessage(), msg text2: "+text);

                //
                // store the message id to use as the Correlator value
                //
                messageID = msg.getJMSMessageID();

                // Call a private method to put the message onto another queue
                putMessage(messageID, text);
        }
        catch  (Exception err)
        {
                err.printStackTrace();
        }
        return;
}
```

What to do next

After you have developed an enterprise application to use message-driven beans, configure and deploy the application. For example, define activation specifications for the message-driven beans and, optionally, change the deployment descriptor attributes for the application. For more information, see “Deploying an

enterprise application to use message-driven beans with JCA 1.5-compliant resources" on page 2084 and "Deploying an enterprise application to use message-driven beans with listener ports" on page 2088.

JMS report messages

JMS applications can use report messages as a form of managed request/response processing, to give remote feedback to producers on the outcome of their send operations and the fate of their messages.

JMS applications can request the following types of report message by setting appropriate JMS_IBM_Report_Xxxx message properties and options. The options have the same general syntax and meaning:

MQRO_*report-type*
A report message of the indicated type is generated that contains the MQMD of the original message. It does not contain any message body data.

MQRO_*report-type*_**WITH_DATA**
A report message of the indicated type is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_*report-type*_**WITH_FULL_DATA**
A report message of the indicated type is generated that contains all data from the original message.

Use the following prefix with each option: com.ibm.websphere.sib.api.jms.

For example, to request a Confirm on delivery (COD) report message with full data, the JMS application must set JMS_IBM_Report_COD to the value com.ibm.websphere.sib.api.jms.MQRO_COD_WITH_FULL_DATA.

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Exception	Send a report message if the request message cannot be put to the target queue. The exception report messages are generated when a message has been rerouted to an exception destination.	JMS_IBM_Report_Exception • MQRO_EXCEPTION • MQRO_EXCEPTION_WITH_DATA • MQRO_EXCEPTION_WITH_FULL_DATA
Discard	Discard the original request message rather than sending it to an exception destination. You can use this option with the JMS_IBM_Report_Exception property set to MQRO_EXCEPTION_WITH_FULL_DATA to return an undeliverable request message to its sender.	JMS_IBM_Report_Discard_Msg • TRUE • FALSE
Expiration	Send a report message if the request message passes its expiry time.	JMS_IBM_Report_Expiration • MQRO_EXPIRATION • MQRO_EXPIRATION_WITH_DATA • MQRO_EXPIRATION_WITH_FULL_DATA

Type of report message	Description	JMS_IBM_Report_Xxxx message property and options
Confirm on arrival (COA)	Send a report message when the request message has been put to the target queue. For publish/subscribe messaging, the COA report message is generated only on the producers messaging engine. Therefore, such reports are relevant only to local subscriptions. For point-to-point messaging, COA messages are generated when the message arrives at the final destination. For partitioned queues, the report message is generated only when the put operation has committed and a final destination has therefore been selected. Any With_Data or With_Full_Data report options specified are ignored; the COA report message deals only with message headers. If a forward-routing path is used, the COA message are generated when the message arrives at the final destination in the path.	JMS_IBM_Report_COA • MQRO_COA • MQRO_COA_WITH_DATA • MQRO_COA_WITH_FULL_DATA
Confirm on delivery (COD)	Send a report message when the request message has been removed from the queue or topic space by a message consumer. For publish/subscribe messaging, the COD message is generated when all subscribers have received the request message. Therefore, there is one COD message generated for every COA. When a message is consumed by a subscriber, the reference count of the message on the topic space is reduced. When the reference count reaches zero, the message is removed from the topic space then a COD report message is generated. For point-to-point messaging, the COD message is generated after the message has been successfully received by a consuming application. Any With_Data or With_Full_Data report options specified are ignored; the COD report message deals only with message headers.	JMS_IBM_Report_COD • MQRO_COD • MQRO_COD_WITH_DATA • MQRO_COD_WITH_FULL_DATA
Positive action notification (PAN)	Ask the consumer application to send a report message when it has successfully processed the request message.	JMS_IBM_Report_PAN • TRUE • FALSE
Negative action notification (NAN)	Ask the consumer application to send a report message if it has not successfully processed the request message.	JMS_IBM_Report_NAN • TRUE • FALSE

The requesting application can control other aspects of the report message as follows:

- How the message Id is generated for the report message and any reply message:

MQRO_New_Msg_Id
This the default. A new message Id is generated for the report message.

MQRO_Pass_Msg_Id
The message Id of the report message is set to the message Id of the request message.

- How the correlation Id of the report or reply message is to be set.

MQRO_Copy_Msg_Id_To_Correl_Id
This the default. the correlation Id of the report message is set to the message Id of the request message.

MQRO_Pass_Correl_Id
The correlation Id of the report message is set to the correlation Id of the request message.

For more information about report messages and the associated properties and options, see the *Report messages* section of the WebSphere MQ information center.

JMS interfaces

WebSphere Application Server supports applications that use JMS 1.1 domain-independent interfaces (referred to as “common interfaces” in the JMS specification) and JMS 1.0.2 domain-specific interfaces.

With JMS 1.1, the preferred approach for implementing applications is to use common interfaces because they provide a simpler programming model than domain-specific interfaces. Also, applications can create both queues and topics in the same session and coordinate their use in the same transaction. Common interfaces are parents of domain-specific interfaces.

Note: Domain-specific interfaces (provided for JMS 1.0.2 in WebSphere Application Server Version 5) are supported only to provide compatibility for applications that have already been implemented to use those interfaces.

Table 82. The point-to-point and publish/subscribe versions of JMS common interfaces. The first column of the table lists the JMS common interfaces. The second column lists the point-to-point versions of the JMS common interfaces. The third column lists the publish/subscribe versions of the JMS common interfaces.

JMS common interfaces	Point-to-point interfaces	Publish/subscribe interfaces
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession,
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

For more information about JMS interfaces, see the JMS documentation at http://java.sun.com/products/jms/docs.html.

Chapter 14. Developing Naming and directory

This page provides a starting point for finding information about naming support. Naming includes both server-side and client-side components. The server-side component is a Common Object Request Broker Architecture (CORBA) naming service (CosNaming). The client-side component is a Java Naming and Directory Interface (JNDI) service provider. JNDI is a core component in the Java Platform, Enterprise Edition (Java EE) programming model.

The WebSphere JNDI service provider can be used to interoperate with any CosNaming name server implementation. Yet WebSphere name servers implement an extension to CosNaming, and the JNDI service provider uses those WebSphere extensions to provide greater capability than CosNaming alone. Some added capabilities are binding and looking up of non-CORBA objects.

Java EE applications use the JNDI service provider supported by WebSphere Application Server to obtain references to objects related to server applications, such as enterprise bean (EJB) homes, which have been bound into a CosNaming name space.

Developing applications that use JNDI

References to enterprise bean (EJB) homes and other artifacts such as data sources are bound to the WebSphere Application Server name space. These objects can be obtained through Java Naming and Directory Interface (JNDI). Before you can perform any JNDI operations, you need to get an initial context. You can use the initial context to look up objects bound to the name space.

About this task

The following examples describe how to get an initial context and how to perform lookup operations.
- Getting the default initial context
- Getting an initial context by setting the provider URL property
- Setting the provider URL property to select a different root context as the initial context
- Looking up an EJB home with JNDI

In these examples, the default behavior of features specific to the WebSphere Application Server JNDI Context implementation is used.

The WebSphere Application Server JNDI context implementation includes special features. JNDI caching enhances performance of repeated lookup operations on the same objects. Name syntax options offer a choice of a name syntaxes, one optimized for typical JNDI clients, and one optimized for interoperability with CosNaming applications. Most of the time, the default behavior of these features is the preferred behavior. However, sometimes you should modify the behavior for specific situations.

JNDI caching and name syntax options are associated with a javax.naming.InitialContext instance. To select options for these features, set properties that are recognized by the WebSphere Application Server initial context factory. To set JNDI caching or name syntax properties which will be visible to the initial context factory, do the following:

Procedure

1. Optional: Configure JNDI caches

   JNDI caching can greatly increase performance of JNDI lookup operations. By default, JNDI caching is enabled. In most situations, this default is the desired behavior. However, in specific situations, use the other JNDI cache options.

Objects are cached locally as they are looked up. Subsequent lookups on cached objects are resolved locally. However, cache contents can become stale. This situation is not usually a problem, since most objects you look up do not change frequently. If you need to look up objects which change relatively frequently, change your JNDI cache options.

JNDI clients can use several properties to control cache behavior.

You can set properties:

- From the command line by entering the actual string value. For example:

  ```
  java -Dcom.ibm.websphere.naming.jndicache.maxentrylife=1440
  ```

- In a `jndi.properties` file by creating a file named `jndi.properties` as a text file with the desired properties settings. For example:

  ```
  ...
  com.ibm.websphere.naming.jndicache.cacheobject=none
  ...
  ```

  If you use this technique, be aware that other instances of the `jndi.properties` file might exist in the classpath, and might contain conflicting property settings. Property settings are determined by the order in which the class loader picks up the `jndi.properties` files. There is no way to control the order that the class loader uses to locate files in the classpath. WebSphere Application Server does not initially contain or create any `jndi.properties` files that set the com.ibm.websphere.naming.jndicache.cacheobject property.

- Within a Java program by using the PROPS.JNDI_CACHE* Java constants, defined in the `com.ibm.websphere.naming.PROPS` file. The constant definitions follow:

  ```
  public static final String JNDI_CACHE_OBJECT =
   "com.ibm.websphere.naming.jndicache.cacheobject";
  public static final String JNDI_CACHE_OBJECT_NONE      = "none";
  public static final String JNDI_CACHE_OBJECT_POPULATED = "populated";
  public static final String JNDI_CACHE_OBJECT_CLEARED   = "cleared";
  public static final String JNDI_CACHE_OBJECT_DEFAULT   =
   JNDI_CACHE_OBJECT_POPULATED;

  public static final String JNDI_CACHE_NAME =
   "com.ibm.websphere.naming.jndicache.cachename";
  public static final String JNDI_CACHE_NAME_DEFAULT = "providerURL";

  public static final String JNDI_CACHE_MAX_LIFE =
   "com.ibm.websphere.naming.jndicache.maxcachelife";
  public static final int    JNDI_CACHE_MAX_LIFE_DEFAULT = 0;

  public static final String JNDI_CACHE_MAX_ENTRY_LIFE =
   "com.ibm.websphere.naming.jndicache.maxentrylife";
  public static final int    JNDI_CACHE_MAX_ENTRY_LIFE_DEFAULT = 0;
  ```

  To use the previous properties in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor as follows:

  ```
  java.util.Hashtable env = new java.util.Hashtable();
  ...

  // Disable caching
  env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE); ...
  javax.naming.Context initialContext = new javax.naming.InitialContext(env);
  ```

Following are examples that illustrate how you can use JNDI cache properties to achieve the desired cache behavior. Cache properties take effect when an InitialContext object is constructed.

Example: Controlling JNDI cache behavior from a program

```
import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.naming.Context;
import com.ibm.websphere.naming.PROPS;

/*****
 Caching discussed in this section pertains to the WebSphere Application Server initial context factory.
 Assume the property, java.naming.factory.initial, is set to
 "com.ibm.websphere.naming.WsnInitialContextFactory" as a java.lang.System property.
*****/

Hashtable env;
```

```
Context ctx;

// To clear a cache:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_CLEARED);
ctx = new InitialContext(env);

// To set a cache's maximum cache lifetime to 60 minutes:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_MAX_LIFE, "60");
ctx = new InitialContext(env);

// To turn caching off:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
ctx = new InitialContext(env);

// To use caching and no caching:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_POPULATED);
ctx = new InitialContext(env);
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
Context noCacheCtx = new InitialContext(env);

Object o;

// Use caching to look up home, since the home should rarely change.
o = ctx.lookup("com/mycom/MyEJBHome");
// Narrow, etc. ...

// Do not use cache if data is volatile.
o = noCacheCtx.lookup("com/mycom/VolatileObject");
// ...
```

Example: Looking up a JavaMail session with JNDI

The following example shows a lookup of a JavaMail resource:

```
// Get the initial context as shown above
...
Session session =
     (Session) initialContext.lookup("java:comp/env/mail/MailSession");
```

2. Optional: Specify the name syntax

   INS syntax is designed for JNDI clients that need to interoperate with CORBA applications. This syntax allows a JNDI client to make the proper mapping to and from a CORBA name. INS syntax is very similar to the JNDI syntax with the additional special character, dot (.). Dots are used to delimit the id and kind fields in a name component. A dot is interpreted literally when it is escaped. Only one unescaped dot is allowed in a name component. A name component with a non-empty id field and empty kind field is represented with only the id field value and must not end with an unescaped dot. An empty name component (empty id and empty kind field) is represented with a single unescaped dot. An empty string is not a valid name component representation.

   JNDI name syntax is the default syntax and is suitable for typical JNDI clients. This syntax includes the following special characters: forward slash (/) and backslash (\). Components in a name are delimited by a forward slash. The backslash is used as the escape character. A forward slash is interpreted literally if it is escaped, that is, preceded by a backslash. Similarly, a backslash is interpreted literally if it is escaped.

   Most WebSphere applications use JNDI to look up EJB objects and do not need to look up objects bound by CORBA applications. Therefore, the default name syntax used for JNDI names is the most convenient. If your application needs to look up objects bound by CORBA applications, you may need to change your name syntax so that all CORBA CosNaming names can be represented.

   JNDI clients can set the name syntax by setting a property. The property setting is applied by the initial context factory when you instantiate a new java.naming.InitialContext object. Names specified in JNDI operations on the initial context are parsed according to the specified name syntax.

You can set the property:

- From a command line, enter the actual string value. For example:

```
java -Dcom.ibm.websphere.naming.name.syntax=ins
```

- Create a file named `jndi.properties` as a text file with the desired properties settings. For example:

```
...
com.ibm.websphere.naming.name.syntax=ins
...
```

  If you use this technique, be aware that other instances of the `jndi.properties` file might exist in the classpath, and might contain conflicting property settings. Property settings are determined by the order in which the class loader picks up the `jndi.properties` files. There is no way to control the order that the class loader uses to locate files in the classpath. WebSphere Application Server does not initially contain or create any `jndi.properties` files that set the com.ibm.websphere.naming.name.syntax property.

- Use the PROPS.NAME_SYNTAX* Java constants, defined in the `com.ibm.websphere.naming.PROPS` file, in a Java program. The constant definitions follow:

```
public static final String NAME_SYNTAX =
    "com.ibm.websphere.naming.name.syntax";
public static final String NAME_SYNTAX_JNDI = "jndi";
public static final String NAME_SYNTAX_INS  = "ins";
```

  To use the previous properties in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor as follows:

```
java.util.Hashtable env = new java.util.Hashtable();
...
env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS); // Set name syntax to INS
...
javax.naming.Context initialContext = new javax.naming.InitialContext(env);
```

Example: Setting the syntax used to parse name strings

The name syntax property can be passed to the InitialContext constructor through its parameter, in the System properties, or in a `jndi.properties` file. The initial context and any contexts looked up from that initial context parse name strings based on the specified syntax.

The following example shows how to set the name syntax to make the initial context parse name strings according to INS syntax.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS; // WebSphere naming constants
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, ...);
env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS);
Context initialContext = new InitialContext(env);
// The following name maps to a CORBA name component as follows:
//    id = "a.name", kind = "in.INS.format"
// The unescaped dot is used as the delimiter.
// Escaped dots are interpreted literally.
java.lang.Object o = initialContext.lookup("a\\.name.in\\.INS\\.format");
...
```

INS name syntax requires that embedded periods (.) in a name such as `in.INS.format` be escaped using a backslash character (\). In a Java String literal, a backslash character (\) must be escaped with another backslash character (\\).

3. Optional: Disable host name normalization

   References to host names, IP addresses, and `localhost` in provider URLs are typically normalized. The format of a normalized host name is the fully-qualified form of the host name. Host name normalization improves system efficiency because it enables the same JNDI cache to be used for a given bootstrap host regardless of the format of the reference in the provider URL. For example, host name normalization enables the same JNDI cache to be used for `myhost`, `myhost.mydomain.com`, and `localhost` references if all of these references refer to the same host.

Because normalized host names are cached, subsequent normalizations execute more quickly. In some network environments, domain name lookup data changes dynamically, causing the cached host name normalization data to become stale. In such environments, you might need to disable hostname normalization. When you disable host normalization, host name and IP addresses are used as is. References to `localhost` typically resolve to the loopback address, 127.0.0.1.

JNDI clients can disable host name normalization by setting a property. The property setting is applied by the initial context factory when you instantiate a new java.naming.InitialContext object.

Use one of the following techniques to set this property:

- You can enter the actual string value from a command line. For example:

  ```
  java -Dcom.ibm.websphere.naming.hostname.normalizer=...none...
  ```

- You can create a file named `jndi.properties` as a text file with the desired properties settings. For example:

  ```
  ...
  com.ibm.websphere.naming.hostname.normalizer=...none...
  ...
  ```

  If you use this technique, be aware that other instances of the `jndi.properties` file might exist in the classpath, and might contain conflicting property settings. Property settings are determined by the order in which the class loader picks up the `jndi.properties` files. There is no way to control the order that the class loader uses to locate files in the classpath. WebSphere Application Server does not initially contain or create any `jndi.properties` files that set the com.ibm.websphere.naming.hostname.normalizer property.

- You can use the PROPS.HOSTNAME_NORMALIZER* Java constants in a Java program. These Java constants are defined in the com.ibm.websphere.naming.PROPS file. Following are the constant definitions to specify if you use this technique:

  ```
  public static final String HOSTNAME_NORMALIZER =
      "com.ibm.websphere.naming.hostname.normalizer";
  public static final String HOSTNAME_NORMALIZER_NONE = "...none...;
  ```

  To use these definitions in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor:

  ```
  java.util.Hashtable env = new java.util.Hashtable();
  ...
  env.put(PROPS.HOSTNAME_NORMALIZER, PROPS.HOSTNAME_NORMALIZER_NONE);
      // Disable hostname normalization
  ...
  javax.naming.Context initialContext =
     new javax.naming.InitialContext(env);
  ```

Example: Disabling host name normalization

You can pass the host name normalizer property to the InitialContext constructor through the InitialContext constructor parameter in the system properties file, or in a `jndi.properties` file. The initial context and any future contexts looked up from that initial context use this property setting.

The following example shows how to disable host name normalization.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS; // WebSphere naming constants
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, ...);
env.put(PROPS.HOSTNAME_NORMALIZER, PROPS.HOSTNAME_NORMALIZER_NONE);
Context initialContext = new InitialContext(env);
java.lang.Object o = initialContext.lookup(...);
...
```

Example: Getting the default initial context

There are various ways for a program to get the default initial context.

The following example gets the default initial context. Note that no provider URL is passed to the javax.naming.InitialContext constructor.

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
...
Context initialContext = new InitialContext();
...
```

The default initial context returned depends the runtime environment of the Java Naming and Directory Interface (JNDI) client. Following are the initial contexts returned in the various environments:

Thin client
The initial context is the server root context of the server running on the local host at port 2809.

Pure client
The initial context is the context specified by the java.naming.provider.url property passed to launchClient command with the -CCD command line parameter. The context usually is the server root context of the server at the address specified in the URL, although it is possible to construct a corbaname or corbaloc URL which resolves to some other context.

If no provider URL is specified, it is the server root context of the server running on the host and port specified by the `-CCproviderURL`, or `-CCBootstrapHost` and `-CCBootstrapPort` command line parameters. The default host is the local host, and the default port is 2809.

Server process
The initial context is the server root context for that process.

Even though no provider URL is explicitly specified in the previous example, the InitialContext constructor might find a provider URL defined in other places that it searches for property settings.

Users of properties which affect ORB initialization should read the rest of this section for a deeper understanding of exactly how initial contexts are obtained.

Determining which server is used to obtain the initial context

WebSphere Application Server name servers are CORBA CosNaming name servers, and the product provides a CosNaming JNDI plug-in implementation for JNDI clients to perform naming operations on product namespaces. The CosNaming plug-in implementation is selected through a JNDI property that is passed to the InitialContext constructor. This property is `java.naming.factory.initial`, and it specifies the initial context factory implementation to use to obtain an initial context. The factory returns a javax.naming.Context instance, which is part of its implementation.

The initial context factory, `com.ibm.websphere.naming.WsnInitialContextFactory`, is typically used by applications to perform JNDI operations. The WebSphere Application Server runtime environment is set up to use this initial context factory if one is not specified explicitly by the JNDI client. When the initial context factory is invoked, an initial context is obtained. The following paragraphs explain how the initial context factory obtains the initial context in client and server environments.

- Registration of initial references in server processes

  Every WebSphere Application Server has an ORB used to receive and dispatch invocations on objects running in that server. Services running in the server process can register initial references with the ORB. Each initial reference is registered under a key, which is a string value. An initial reference can be any CORBA object. WebSphere Application Server name servers register several initial contexts as initial references under predefined keys. Each name server initial reference is an instance of the interface org.omg.CosNaming.NamingContext.
- Obtaining initial references in pure client processes

  Pure JNDI clients, that is, JNDI clients which are not running in a WebSphere Application Server process, also have an ORB instance. This client ORB instance can be passed to the InitialContext

constructor, but typically the initial context factory creates and initializes the client ORB instance transparently. A client ORB can be initialized with initial references, but the initial references most likely resolve to objects running in some server. The initial context factory does not define any default initial references when it initializes an ORB. If the `resolve_initial_references` method is invoked on the client ORB when no initial references have been configured, the method invocation fails. This condition is typical for pure client processes. To obtain an initial NamingContext reference, the initial context factory must invoke `string_to_object` with an IIOP type CORBA object URL, such as `corbaloc:iiop:myhost:2809`. The URL specifies the address of the server from which to obtain the initial context. The host and port information is extracted from the provider URL passed to the InitialContext constructor.

If no provider URL is defined, the WebSphere Application Server initial context factory uses the default provider URL of `corbaloc:iiop:localhost:2809`.

The `string_to_object` ORB method resolves the URL and communicates with the target server ORB to obtain the initial reference.

- Obtaining initial references in server processes

  If the JNDI client is running in a WebSphere Application Server process, the initial context factory obtains a reference to the server ORB instance if the JNDI client does not provide an ORB instance. Typically, JNDI clients running in server processes use the server ORB instance; that is, they do not pass an ORB instance to the InitialContext constructor. The name server which is running in the server process sets a provider URL as a java.lang.System property to serve as the default provider URL for all JNDI clients in the process. This default provider URL is `corbaloc:rir:/NameServiceServerRoot`. This URL resolves to the server root context for that server. (The URL is equivalent to invoking `resolve_initial_references` on the ORB with a key of NameServiceServerRoot. The name server registers the server root context as an initial reference under that key.)

- Understanding the legacy ORB protocol

  Releases previous to WebSphere Application Server Version 5 used a different ORB implementation, which used a legacy protocol in contrast with the Interoperable Name Service (INS) protocol now used. This change has affected the implementation of the initial context factory. Certain types of pure clients can experience different behavior when getting initial JNDI contexts as compared to previous releases of WebSphere Application Server. This behavior is discussed in more detail later in this section.

  The following ORB properties are used with the legacy ORB protocol for ORB initialization and are now deprecated:
  - com.ibm.CORBA.BootstrapHost
  - com.ibm.CORBA.BootstrapPort

  The new INS ORB is different in a major respect, in that it exhibits no default behavior if no initial references are defined.

  In the legacy ORB, the bootstrap host and port values defaulted to `localhost` and `900`.

  All initial references were obtained from the server running on the bootstrap host and port. So, if the ORB user provided no bootstrap host and port, all initial references are resolved from the server running on the local host at port 900. The INS ORB has no concept of bootstrap host or bootstrap port. All initial references are defined independently. That is, different initial references could resolve to different servers. If `ORB.resolve_initial_references` is invoked with a key such that the ORB is not initialized with an initial reference having that key, the call fails.

  In releases previous to Version 5, the initial context factory invoked `resolve_initial_references` on the ORB in the absence of any provider URL. This action succeeded if a name server at the default bootstrap host and port was running. In the current release, with the INS ORB, this would fail. (Actually, the ORB would fall back to the legacy protocol during the deprecation period, but when the legacy protocol is no longer supported, the operation would fail.)

  The initial context factory now uses a default provider URL of `corbaloc:iiop:localhost:2809`, and invokes `string_to_object` with the provider URL.

This operation preserves the behavior that pure clients in previous releases experienced when they set no ORB bootstrap properties or provider URL. However, this different initial context factory implementation changes the behavior experienced by certain legacy pure clients, which do not specify a provider URL:
– Clients which set the ORB bootstrap properties previously listed when getting an initial context.
– Clients which supply their own ORB instance to the InitialContext constructor.

There are two ways to circumvent this change of behavior:

– Always specify an IIOP type provider URL. This approach does not depend on the bootstrap host and port properties and continues to work when support for the bootstrap host and port properties is removed. For example, you can express bootstrap host and port property values of `myHost` and `2809`, respectively, as `corbaloc:iiop:myHost:2809`.
– Use an rir type provider URL:
  - Specify `corbaloc:rir:/NameServiceServerRoot` if the ORB is initialized to use a Version 5 server as the bootstrap server.
  - Specify `corbaname:rir:/NameService#domain/legacyRoot` if the ORB is initialized to use a Version 4.0.x server as the bootstrap server.
  - Specify `corbaloc:rir:/NameService` if the ORB is initialized to use a server other than a Version 5 or 4.0.x server as the bootstrap server.

  URLs of this type are equivalent to invoking `resolve_initial_references` on the ORB with the specified key. If the bootstrap host and port properties are being used to initialize the ORB, this approach will not work when the bootstrap and host properties are no longer supported.

• The InitialContext constructor search order for JNDI properties

  If the code snippet shown at the beginning of this section is executed by an application, the bootstrap server depends on the value of the property, java.naming.provider.url.

  If the property is not set (in server processes the default value is set as a system property), the default host of `localhost` and default port of `2809` are used as the address of the server from which to obtain the initial context.

  The JNDI specification describes where the InitialContext constructor looks for java.naming.provider.url property settings, but briefly, the property is picked up from the following places in the order shown:

  InitialContext constructor
  This does not apply to the previous example because the example uses the empty InitalContext constructor.

  System environment
  You can add JNDI properties to the system environment as an option on the Java command invocation and by program code. The recommended way to set the provider URL in the system environment is as an option supplied to the Java command invocation. Setting the provider URL in this manner is not temporal, so that getting a default initial context will always yield the same result. It is generally recommended that program code not set the provider URL property in the system environment because as a side-effect, this could adversely affect other, possibly unrelated, code running elsewhere in the same process.

  `jndi.properties` file
  There may be many `jndi.properties` files that are within the scope of the class loader in effect. All `jndi.properties` files are used for setting JNDI properties, but the provider URL setting is determined by the first `jndi.properties` file returned by the class loader.

Example: Getting an initial context by setting the provider URL property

In general, Java Naming and Directory Interface (JNDI) clients should assume the correct environment is already configured so there is no need to explicitly set property values and pass them to the `InitialContext` constructor. However, a JNDI client might need to access a namespace other than the one identified in its environment. In this case, it is necessary to explicitly set the java.naming.provider.url (provider URL) property used by the `InitialContext` constructor. A provider URL contains bootstrap server

information that the initial context factory can use to obtain an initial context. Any property values passed in directly to the `InitialContext` constructor take precedence over settings of those same properties found elsewhere in the environment.

You can use two different provider URL forms with theWebSphere Application Server initial context factory:
- A CORBA object URL
- An IIOP URL

CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. CORBA object URLs are part of the OMG CosNaming Interoperable Naming Specification. A corbaname URL, for example, can include initial context and lookup name information and can be used as a lookup name without the need to explicitly obtain another initial context. The IIOP URLs are the legacy JNDI format, but are still supported by the WebSphere Application Server initial context factory.

The following examples illustrate the use of these URLs.
- "Using a CORBA object URL"
- "Using a CORBA object URL with multiple name server addresses"
- "Using a CORBA object URL from a non-WebSphere Application Server JNDI implementation" on page 634
- "Using an IIOP URL" on page 634

Using a CORBA object URL

This example shows a CORBA object URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Using a CORBA object URL with multiple name server addresses

CORBA object URLs can contain more than one bootstrap address. You can use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap addresses for all servers in the cluster in the URL. The operation succeeds if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address list will be processed. For example, the second bootstrap address may be used to obtain the initial context even though the server at the first bootstrap address in the list is available.

Multiple-address provider URLs resolving to servers on non-z/OS systems cannot contain bootstrap addresses for node agent processes. This is because bootstrapping to a nodeagent returns a non-WLM-enabled initial context. As a result, WLM is not used. Instead, the URLs should contain only bootstrap addresses of members of the same cluster so that the initial context is WLM-enabled. If the URLs contain the bootstrap addresses of multiple clusters, incorrect behavior could result. A given name might not resolve, or resolve to the same object on all clusters. When resolving to servers running on the z/OS operating system, the URLs can contain bootstrap addresses for node agent processes since an z/OS a node agent does return a WLM-enabled initial context.

An example of a corbaloc URL with multiple addresses follows.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
// All of the servers in the provider URL below are members of
// the same cluster.
env.put(Context.PROVIDER_URL,
     "corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810");
Context initialContext = new InitialContext(env);
...
```

Using a CORBA object URL from a non-WebSphere Application Server JNDI implementation

Initial context factories for CosNaming JNDI plug-in implementations other than the WebSphere Application Server initial context factory most likely obtain an initial context using the object key, `NameService`. When you use such a context factory to obtain an initial context from a WebSphere Application Server name server, the initial context is the cell root context. Since system artifacts such as EJB homes associated with a server are bound under the server's server root context, names used in JNDI operations must be qualified. If you want to use relative names, ensure your initial context is the server root context under which the target object is bound. In order to make the server root context the initial context, specify a corbaloc provider URL with an object key of `NameServiceServerRoot`.

This example shows a CORBA object type URL from a non-WebSphere Application Server JNDI implementation. This example assumes full CORBA object URL support by the non-WebSphere Application Server JNDI implementation. The object key of `NameServiceServerRoot` is specified so that the initial context will be the specified server's server root context.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.somecompany.naming.TheirInitialContextFactory");
env.put(Context.PROVIDER_URL,
      "corbaname:iiop:myhost.mycompany.com:9810/NameServiceServerRoot");
Context initialContext = new InitialContext(env);
...
```

If qualified names are used, you can use the default key of `NameService`.

Using an IIOP URL

The IIOP type of URL is a legacy format which is not as flexible as CORBA object URLs. However, URLs of this type are still supported. The following example shows an IIOP type URL as the provider URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "iiop://myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Example: Setting the provider URL property to select a different root context as the initial context

Each server contains its own server root context, and, when bootstrapping to a server, the server root is the default initial JNDI context. Most of the time, this default is the desired initial context, since system

artifacts such as EJB homes are bound there. However, other root contexts exist, which can contain bindings of interest. It is possible to specify a provider URL to select other root contexts.

Examples for selecting other root contexts follow:
- Initial root contexts with a CORBA object URL
- Initial root contexts with the namespace root property

Selecting the initial root context with a CORBA object URL

There are several object keys registered with the bootstrap server that you can use to select the root context for the initial context. To select a particular root context with a CORBA object URL object key, set the object key to the corresponding value. The default object key is NameService. Using JNDI yields the server root context. A table that lists the different root contexts and their corresponding object key follows:

Root Context	CORBA Object URL Object Key
Server Root	NameServiceServerRoot
Cell Persistent Root	NameServiceCellPersistentRoot
Cell Root	NameServiceCellRoot
Node Root	NameServiceNodeRoot

The following example shows the use of a corbaloc URL with the object key set to select the cell persistent root context as the initial context.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL,
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceCellPersistentRoot");
Context initialContext = new InitialContext(env);
...
```

Selecting the initial root context with the namespace root property

You can also select the initial root context by passing a namespace root property setting to the InitialContext constructor. Generally, the object key setting previously described is sufficient. Sometimes a property setting is preferable. For example, you can set the root context property on the Java invocation to make which server root is being used as the initial context transparent to the application. The default server root property setting is `defaultroot`, which yields the server root context.

Root Context	Namespace Root Property Value
Server Root	bootstrapserverroot
Cell Persistent Root	cellpersistentroot
Cell Root	cellroot
Node Root	bootstrapnoderoot

The initial context factory ignores the namespace root property if the provider URL contains an object key other than `NameService`.

The following example shows use of the namespace root property to select the cell persistent root context as the initial context. Note that available constants are used instead of hard-coding the property name and value.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
env.put(PROPS.NAME_SPACE_ROOT, PROPS.NAME_SPACE_ROOT_CELL_PERSISTENT);
Context initialContext = new InitialContext(env);
...
```

Example: Looking up an EJB home or business interface with JNDI

Most applications that use Java Naming and Directory Interface (JNDI) run in a container. Some do not. The name used to look up an object depends on whether or not the application is running in a container. Sometimes it is more convenient for an application to use a corbaname URL as the lookup name. Container-based JNDI clients and thin Java clients can use a corbaname URL.

The following examples show how to perform JNDI lookups from different types of applications.

- "JNDI lookup from an application running in a container"
- "JNDI lookup from an application that does not run in a container" on page 637
- "JNDI lookup with a corbaname URL" on page 639

JNDI lookup from an application running in a container

Applications that run in a container can use `java:` lookup names. Lookup names of this form provide a level of indirection such that the lookup name used to look up an object is not dependent on the object's name as it is bound in the name server's namespace. The deployment descriptors for the application provide the mapping from the `java:` name and the name server lookup name. The container sets up the `java:` namespace based on the deployment descriptor information so that the `java:` name is correctly mapped to the corresponding object.

The following example shows a lookup of an EJB 3.0 remote business interface. The actual home lookup name is determined by the interface's `ibm-ejb-jar-bnd.xml` binding file, if present, or by the default name assigned by the EJB container if no binding file is present. For more information, see topics on default bindings for business interfaces and homes and on user-defined bindings for EJB business interfaces and homes.

```
// Get the initial context as shown in a previous example.
...
// Look up the business interface using the JNDI name.
try {
   java.lang.Object ejbBusIntf =
      initialContext.lookup(
         "java:comp/env/com/mycompany/accounting/Account");
   accountIntf =
      (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
   catch (NamingException e) { // Error getting the business interface
   ...
}
```

The following example shows a lookup of an EJB 1.x or 2.x EJB home. The actual home lookup name is determined by the application's deployment descriptors. The enterprise bean (EJB) resides in an EJB container, which provides an interface between the bean and the application server on which it resides.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome =
```

```
      initialContext.lookup(
         "java:comp/env/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
   catch (NamingException e) { // Error getting the home interface
   ...
}
```

JNDI lookup from an application that does not run in a container

Applications that do not run in a container cannot use `java:` lookup names because it is the container which sets the `java:` namespace up for the application. Instead, an application of this type must look the object up directly from the name server. Each application server contains a name server. System artifacts such as EJB homes are bound relative to the server root context in that name server. The various name servers are federated by means of a system namespace structure. The recommended way to look up objects on different servers is to qualify the name so that the name resolves from any initial context in the cell. If a relative name is used, the initial context must be the same server root context as the one under which the object is bound. The form of the qualified name depends on whether the qualified name is a topology-based name or a fixed name. Examples of each form of qualified name follow.

- **Topology-based qualified names**

  Topology-based qualified names traverse through the system namespace to the server root context under which the target object is bound. A topology-based qualified name resolves from any initial context in the cell.

  The topology-based qualified name depends on whether the object resides on a single server or server cluster. Examples of each lookup follow.

  Single server

  The following example shows a lookup of an EJB business interface that is running in the single server, MyServer, configured in the node, Node1.

  ```
  // Get the initial context as shown in a previous example.
  // Using the form of lookup name below, it does not matter which
  // server in the cell is used to obtain the initial context.
  ...
  // Look up the business interface using the JNDI name
  try {
     java.lang.Object ejbBusIntf = initialContext.lookup(
        "cell/nodes/Node1/servers/MyServer/com/mycompany/accounting/Account");
     accountIntf =
       (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
  }
  catch (NamingException e) { // Error getting the business interface
     ...
  }
  ```

  The following example shows a lookup of an EJB home that is running in the single server, MyServer, configured in the node, Node1.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "cell/nodes/Node1/servers/MyServer/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

Server cluster

The following example shows a lookup of an EJB business interface which is running in the cluster, MyCluster. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in a previous example.
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
   // Look up the business interface using the JNDI name
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
      "cell/clusters/MyCluster/com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
catch (NamingException e) { // Error getting the business interface
   ...
}
```

The following example shows a lookup of an EJB home which is running in the cluster, MyCluster. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
   // Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "cell/clusters/MyCluster/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

- **Fixed qualified names**

  If the target object has a cell-scoped fixed name defined for it, you can use its qualified form instead of the topology-based qualified name. Even though the topology-based name works, the fixed name does not change with the specific cell topology or with the movement of the target object to a different server.

  An example lookup of an EJB business interface with a qualified fixed name follows.

```
// Get the initial context as shown in a previous example.
// Using the form of lookup name below, it does not matter which
// server in the cell is used to obtain the initial context.
...
// Look up the business interface using the JNDI name
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
     "cell/persistent/com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
   }
catch (NamingException e) { // Error getting the business interface
...

}
```

  An example lookup with a qualified fixed name follows:

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
     "cell/persistent/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
   }
catch (NamingException e) { // Error getting the home interface
...
}
```

JNDI lookup with a corbaname URL

A corbaname can be useful at times as a lookup name. If, for example, the target object is not a member of the federated namespace and cannot be located with a qualifiied name, a corbaname can be a convenient way to look up the object.

A lookup of an EJB business interface with a corbaname URL follows.

```
// Get the initial context as shown in a previous example.
...
// Look up the business interface using a corbaname URL.
try {
   java.lang.Object ejbBusIntf = initialContext.lookup(
      "corbaname:iiop:someHost:2809#com/mycompany/accounting/Account");
   accountIntf =
     (Account)javax.rmi.PortableRemoteObject.narrow(ejbBusIntf, Account.class);
}
catch (NamingException e) { // Error getting the business interface
   ...
}
```

A lookup with a corbaname URL follows.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using a corbaname URL
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "corbaname:iiop:someHost:2809#com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

JNDI interoperability considerations

You must take extra steps to enable your programs to interoperate with non-product JNDI clients and to bind resources from MQSeries® to a namespace.

EJB clients running in an environment other than WebSphere Application Server accessing EJB applications running on product servers

When an enterprise bean (EJB) application running in WebSphere Application Server is accessed by a non-product EJB client, the JNDI initial context factory is presumed to be a non-product implementation. In this case, the default initial context is the cell root. If the JNDI service provider being used supports CORBA object URLs, the corbaname format can be used to look up the EJB home.

The construction of the stringified name depends on whether the object is installed on a single server or cluster.

Single server

Following is a URL that has the bootstrap host `myHost`, the port `2809`, and the enterprise bean installed in the server `server1` in node `node1` and bound in that server under the name `myEJB`:

```
initialContext.lookup(
  "corbaname:iiop:myHost:2809#cell/nodes/node1/servers/server1/myEJB");
```

Server cluster

Following is a URL that has the bootstrap host `myHost`, the port `2809`, and the enterprise bean installed in a server cluster named `myCluster` and bound in that cluster under the name `myEJB`:

```
initialContext.lookup(
    "corbaname:iiop:myHost:2809#cell/clusters/myCluster/myEJB");
```

The lookup works with any name server bootstrap host and port configured in the same cell.

The lookup also works if the bootstrap host and port belong to a member of the cluster itself. To avoid a single point of failure, the bootstrap server host and port for each cluster member can be listed in the URL as follows:

```
initialContext.lookup(
    "corbaname:iiop:host1:9810,:host2:9810#cell/clusters/myCluster/myEJB");
```

The name prefix `cell/clusters/myCluster/` is not necessary if boostrapping to the cluster itself, but it will work. The prefix is needed, however, when looking up enterprise beans in other clusters. Name bindings under the `clusters` context are implemented on the name server to resolve to the server root of a running cluster member during a lookup; thus avoiding a single point of failure.

Without CORBA object URL support

If the JNDI initial context factory being used does not support CORBA object URLs, the initial context can be obtained from the server, and the lookup can be performed on the initial context as follows:

```
Hashtable env = new Hashtable();
env.put(CONTEXT.PROVIDER_URL, "iiop://myHost:2809");
Context ic = new InitialContext(env);
Object o = ic.lookup("cell/clusters/myCluster/myEJB");
```

Binding resources from MQSeries 5.2

In releases previous to WebSphere Application Server Version 5.0, the MQSeries jmsadmin tool could be used to bind resources to the namespace. When used with a WebSphere Application Server namespace, the resource is bound within a transient partition in the namespace and does not persist past the life of the server process. Instead of binding the MQSeries resources with the jmsadmin tool, bind them from the administrative console, under **Resources** in the console navigation tree.

JNDI caching

To increase the performance of Java Naming and Directory Interface (JNDI) operations, the product JNDI implementation employs caching to reduce the number of remote calls to the name server for lookup operations. For most cases, use the default cache setting.

When an InitialContext object is instantiated, an association is established between the InitialContext instance and a cache. The initial context and any contexts returned directly or indirectly from a lookup on the initial context are all associated with that same cache instance. By default, the association is based on the provider URL, in particular, the host name and port. The caller can specify the cache name to override this default behavior. A cache instance of a given name is shared by all instances of InitialContext configured to use a cache of that name which were created with the same context class loader in effect. Two enterprise bean (EJB) applications running in the same server will use their own cache instances, if they are using different context class loaders, even if the cache names are the same.

After an association between an InitialContext instance and cache is established, the association does not change. A `javax.naming.Context` object returned from a lookup operation inherits the cache association of the Context object on which the lookup was performed. Changing cache property values with the `Context.addToEnvironment()` or `Context.removeFromEnvironment()` method does not affect cache behavior. You can change properties affecting a given cache instance with each InitialContext instantiation.

A cache is restricted to a process and does not persist past the life of that process. A cached object is returned from lookup operations until either the maximum cache life for the cache is reached, or the maximum entry life for the object's cache entry is reached. After this time, a lookup on the object causes the cache entry for the object to be refreshed. By default, caches and cache entries have unlimited lifetimes.

Usually, cached objects are relatively static entities, and objects becoming stale is not a problem. However, you can set timeout values on cache entries or on a cache so that cache contents are periodically refreshed.

If a bind or rebind operation is executed on an object, the change is not reflected in any caches other than the one associated with the context from which the bind or rebind was issued. This scenario is most likely to happen when multiple processes are involved, since different processes do not share the same cache, and context objects in all threads in a process typically share the same cache instance for a given name service provider.

JNDI cache settings

Various Java Naming and Directory Interface (JNDI) cache property settings follow. Ensure that all property values are string values.

com.ibm.websphere.naming.jndicache.cachename

The name of the cache to associate with an initial context instance can be specified with this property.

It is possible to create multiple InitialContext instances, each operating on the namespace of a different name server. By default, objects from each bootstrap address are cached separately, since they each involve independent namespaces and name collisions could occur if they used the same cache. The provider URL specified when the initial context is created by default serves as the basis for the cache name. With this property, a JNDI client can specify a cache name. Valid options for cache names follow:

Valid options	Resulting cache behavior
providerURL (default)	Use the value for java.naming.provider.url property as the basis for the cache name. Cache names are based on the bootstrap host and port specified in the URL. The boostrap host is normalized to a fully qualfied name, if possible. For example, "corbaname:iiop:server1:2809#some/starting/context" and "corbaloc:iiop://server1" are normalized to the same cache name. If no provider URL is specified, a default cache name is used.
Any string	Use the specified string as the cache name. You can use any arbitrary string with a value other than "providerURL" as a cache name.

com.ibm.websphere.naming.jndicache.cacheobject

Turn caching on or off and clear an existing cache with this property.

By default, when an InitialContext is instantiated, it is associated with an existing cache or, if one does not exist, a new one is created. An existing cache is used with its existing contents. In some circumstances, this behavior is not desirable. For example, when objects that are looked up change frequently, they can become stale in the cache. Other options are available. The following table lists these other options along with the corresponding property value.

Valid values	Resulting cache behavior
populated (default)	Use a cache with the specified name. If the cache already exists, leave existing cache entries in the cache; otherwise, create a new cache.
cleared	Use a cache with the specified name. If the cache already exists, clear all cache entries from the cache; otherwise, create a new cache.
none	Do not cache. If this option is specified, the cache name is irrelevant. Therefore, this option will not disable a cache that is already associated with other InitialContext instances. The InitialContext that is instantiated is not associated with any cache.

com.ibm.websphere.naming.jndicache.maxcachelife

Impose a limit to the age of a cache with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to `cleared`. This property enables a JNDI client to set the maximum life of a cache. This property differs from the maxentrylife property in that the entire cache is cleared when the cache lifetime is reached. The following table lists the various maxcachelife values and their affect on cache behavior:

Valid values	Resulting cache behavior
0 (default)	Make the cache lifetime unlimited.
Positive integer	Set the maximum lifetime of the entire cache, in minutes, to the specified value. When the maximum lifetime for the cache is reached, the next attempt to read any entry from the cache causes the cache to be cleared

com.ibm.websphere.naming.jndicache.maxentrylife

Impose a limit to the age of individual cache entries with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to `cleared`. This property enables a JNDI client to set the maximum lifetime of individual cache entries. This property differs from the maxcachelife property in that individual entries are refreshed individually as their maximum lifetime reached. This might avoid any noticeable change in performance that might occur if the whole cache is cleared at once. The following table lists the various maxentrylife values and their effect on cache behavior:

Valid values	Resulting cache behavior
0 (default)	Lifetime of cache entries is unlimited.
Positive integer	Set the maximum lifetime of individual cache entries, in minutes, to the specified value. When the maximum lifetime for an entry is reached, the next attempt to read the entry from the cache causes the individual cache entry to refresh.

JNDI to CORBA name mapping considerations

WebSphere Application Server name servers are an implementation of the CORBA CosNaming interface. The product provides a Java Naming and Directory Interface (JNDI) implementation which you can use to access CosNaming name servers through the JNDI interface. Issues can exist when mapping JNDI name strings to and from CORBA names.

Each component in a CORBA name consists of an `id` and `kind` field, but a JNDI name component consists of no such fields. Each component in a JNDI name is atomic. Typical JNDI clients do not need to make a distinction between the `id` and `kind` fields of a name component, or know how JNDI name strings map to CORBA names. JNDI clients of this sort can use the JNDI syntax described later in this section. When a name is parsed according to JNDI syntax, each name component is mapped to the `id` field of the corresponding CORBA name component. The `kind` field always has an empty value. This basic syntax is the least obtrusive to the JNDI client in that it has the fewest special characters. However, you cannot represent with this syntax a CORBA name with a non-empty `kind` field. This restriction can prevent EJB applications from interoperating with CORBA applications.

Some clients, however must interoperate with CORBA applications which use CORBA names with non-empty `kind` fields. These JNDI clients must make a distinction between `id` and `kind` so that JNDI names are correctly mapped to CORBA names, particularly when the CORBA names contain components with non-empty `kind` fields. Such JNDI clients can use the INS name syntax. With its additional special character, you can use INS to represent any CORBA name. Use of this syntax is not recommended unless it is necessary, because this syntax is more restrictive from the JNDI client's perspective in that the JNDI client must be aware that name components with multiple unescaped dots are syntactically invalid. INS name syntax is part of the OMG CosNaming Interoperable Naming Specification.

Developing applications that use CosNaming (CORBA Naming interface)

CORBA clients can perform naming operations on WebSphere Application Server name servers through the CosNaming interface.

About this task

The following examples show how to obtain an ORB instance and an initial context as well as how to look up an EJB home.

Procedure

1. Get an initial context.
2. Perform desired CosNaming operations.

Example: Getting an initial context with CosNaming

In WebSphere Application Server, an initial context is obtained from a bootstrap server. The address for the bootstrap server consists of a host and port. To get an initial context, you must know the host and port for the server that is used as the bootstrap server.

Obtaining an initial context consists of two basic steps:
1. Obtain an ORB reference.
2. Use an ORB reference to get an initial context. Alternatively, use an existing ORB and invoke string_to_object with a CORBA object URL with multiple name server addresses to get an initial context.

Obtaining an ORB reference

Pure CosNaming clients, that is clients that are not running in a server process, must create and initialize an ORB instance with which to obtain the initial context. CosNaming clients which run in server processes can obtain a reference to the server ORB with a JNDI lookup. The following examples illustrate how to create and initialize a client ORB and how to obtain a server ORB reference.

Creating a client ORB instance

To create an ORB instance, invoke the static method, org.omg.CORBA.ORB.init. The init method requires a property set to the name of the ORB class you want to instantiate. An ORB implementation with the class name com.ibm.CORBA.iiop.ORB is included with the product. The WebSphere Application Server ORB recognizes additional properties with which you can specify initial references.

The basic steps for creating an ORB are as follows:
1. Create a Properties object.
2. Set the ORB class property to the product's ORB class.
3. Set the initial reference properties.
4. Invoke ORB.init, passing in the Properties object.

```
...
import java.util.Properties;
import org.omg.CORBA.ORB;
...
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass","com.ibm.ws390.orb.ORB");
props.put("com.ibm.CORBA.ORBInitRef.NameService",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameService");
props.put("com.ibm.CORBA.ORBInitRef.NameServiceServerRoot",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceServerRoot");
ORB _orb = ORB.init((String[])null, props);
...
```

Obtaining a reference to the server ORB

CosNaming clients which run in a server process can obtain a reference to the server ORB with a JNDI lookup on a java: name, shown as follows:

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
import org.omg.CORBA.ORB;
...
Context initialContext = new InitialContext();
ORB orb = (ORB) initialContext.lookup("java:comp/ORB");
...
```

Using an ORB reference to get an initial naming reference

There are two basic ways to get an initial CosNaming context. Both ways involve an ORB method invocation. The first way is to invoke the resolve_initial_references method on the ORB with an initial reference key. For this call to work, the ORB must be initialized with an initial reference for that key. The other way is to invoke the string_to_object method on the ORB, passing in a CORBA object URL with the host and port of the bootstrap server. The following examples illustrate both approaches.

Invoking resolve_initial_references

Once an ORB reference is obtained, invoke the resolve_initial_references method on the ORB to obtain a reference to the initial context. The following code example invokes resolve_initial_reference on an ORB reference.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Obtain ORB reference as shown in examples earlier in this section
...
org.omg.CORBA.Object obj = _orb.resolve_initial_references("NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Note that the key NameService is passed to the resolve_initial_references method. Other initial context keys are registered in product servers. For example, NameServiceServerRoot can be used to obtain a reference to the server root context in the bootstrap name server. For more information on the initial contexts registered in server ORBs, refer to the topic on initial context support.

Invoking string_to_object with a CORBA object URL

You can use an INS-compliant ORB to obtain an initial context even if the ORB is not initialized with any initial references or bootstrap properties, or if those property settings are for a different server than the name server from which you want to obtain the initial context. To obtain an initial context by explicitly specifying the bootstrap name server, invoke the string_to_object method on the ORB, passing in a CORBA object URL which contains the bootstrap server host and port.

The code in the following example invokes the string_to_object method on an existing ORB reference, passing in a CORBA object URL which identifies the desired initial context.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Obtain ORB reference as shown in examples earlier in this section
...
```

```
org.omg.CORBA.Object obj =
 orb.string_to_object("corbaloc:iiop:myhost.mycompany.com:2809/NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Note that the key NameService is used in the corbaloc URL. Other initial context keys are registered in product servers. For example, you can use NameServiceServerRoot to obtain a reference to the server root context in the bootstrap name server.

Using an existing ORB and invoking string_to_object with a CORBA object URL

CORBA object URLs can contain more than one bootstrap server address. Use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap server addresses for all servers in the cluster in the URL. The operation will succeed if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address list will be processed. For example, the second bootstrap server address may be used to obtain the initial context even though the first bootstrap server in the list is available. An example of a corbaloc URL with multiple addresses follows.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Assume orb is an existing ORB instance
org.omg.CORBA.Object obj = orb.string_to_object(
"corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810/NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Example: Looking up an EJB home with CosNaming

You can look up an EJB home or other CORBA object from a WebSphere Application Server name server through the CORBA CosNaming interface.

You can invoke `resolve` or `resolve_str` on the initial context, or you can invoke `string_to_object` on the ORB. You can use a qualified name so that the name resolves regardless of which name server the lookup is executed on, or use an unqualified name that only resolves from the server root context on the name server that actually contains the object binding. (The qualified name traverses the federated system namespace to the specified server root context.)

Qualified and unqualified names

Each application server contains a name server. System artifacts such as EJB homes are bound in that name server. The various name servers are federated by means of a system namespace structure. The recommended way to look up objects on different servers is to use a qualified name.

A qualified name can be a topology-based name, based on the name of the cluster or single server and node that contains the object.

You can define fixed qualified names for objects. With qualified names, you can look up objects residing on different servers from the same initial context by traversing the system namespace structure. Alternatively, you can use an unqualified name, but an unqualified name will only resolve using the name server associated with the object's application server.

CosNaming.resolve (and resolve_str) vs. ORB.string_to_object

If you have an initial context from any name server in a WebSphere Application Server cell, you can look up any CORBA object with a qualified name. You do not need additional host and port information for the target object's name server.

Alternatively, you can look up an object by invoking `string_to_object` on the ORB, passing in a corbaname URL. Typically, an IIOP type URL is specified, so the bootstrap address information required for an initial context must be contained in the URL. You can use a qualified or unqualified stringified name, but an unqualifed name resolves only if the initial context is from the name server in which the object is bound.

The following examples show CosNaming resolve operations using qualified topology-based lookup names and an unqualified lookup name.

CosNaming resolve operation using a qualified name

The topology-based qualified name for an object depends on whether the object is bound in a single server or a server cluster. Examples of each follow.

Single server

The following example shows the lookup of an EJB home that is running in a single server. The enterprise bean that is being looked up is running in the server, `MyServer`, on the node, `Node1`.

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
  "cell/nodes/Node1/servers/MyServer/mycompany/accounting/AccountEJB");
accountHome =
  (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Server cluster

The following example shows a lookup of an EJB home that is running in a cluster. The enterprise bean being that is looked up is running in the cluster, Cluster1. The name can be resolved if any of the cluster members is running.

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
   "cell/clusters/Cluster1/mycompany/accounting/AccountEJB");
accountHome =
   (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

ORB string_to_object operation using an unqualified stringified name

If the resolve operation is being performed on the name server that contains the object, the system namespace does not need to be traversed, and you can use an unqualified lookup name. Note that this name does not resolve on other name servers. If an unqualified name is provided, the object key must be NameServiceServerRoot so that the correct initial context is selected. If a qualified name is provided, you can use the default key of NameService.

The following example shows a lookup of an EJB home. The enterprise bean that is being looked up is bound on the name server running on the host `myHost` on port `2809`. Note the object key of `NameServiceServerRoot`.

```
// Assume orb is an existing ORB instance
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = orb.string_to_object(
    "corbaname:iiop:myHost:2809/NameServiceServerRoot#mycompany/accounting");
accountHome =
    (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Chapter 15. Developing Object pools

This page provides a starting point for finding information about object pools.

Object pools provide an effective means of improving application performance at run time, by supporting the reuse of multiple instances of objects.

Using object pools

An object pool helps an application avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused.

About this task

Object pools are not meant to be used for pooling JDBC connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To use an object pool, the product administrator must define an *object pool manager* using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Note: The Object pool manager service is only supported from within the EJB container or Web container. Looking up and using a configured object pool manager from a Java 2 Platform Enterprise Edition (J2EE) application client container is not supported.

Procedure

1. Start the administrative console.
2. Click **Resources > Object pool managers**.
3. Specify a **Scope** value and click **New**.
4. Specify the required properties for work manager settings.
   Scope The scope of the configured resource. This value indicates the location for the configuration file.
   Name The name of the object pool manager. This name can be up to 30 ASCII characters long.
   JNDI Name
   The Java Naming and Directory Interface (JNDI) name for the pool manager.
5. [Optional] Specify a **Description** and a **Category** for the object pool manager.

Results

After you have completed these steps, applications can find the object pool manager by doing a JNDI lookup using the specified JNDI name.

Example

The following code illustrates how an application can find an object pool manager object:

```
InitialContext ic = new InitialContext();
ObjectPoolManager opm = (ObjectPoolManager)ic.lookup("java:comp/env/pool");
```

When the application has an ObjectPoolManager, it can cache an object pool for classes of the types it wants to use. The following is an example:

```
ObjectPool arrayListPool = null;
ObjectPool vectorPool = null;
try
```

```
{
 arrayListPool = opm.getPool(ArrayList.class);
 vectorPool = opm.getPool(Vector.class);
}
catch(InstantiationException e)
{
 // problem creating pool
}
catch(IllegalAccessException e)
{
 // problem creating pool
}
```

When the application has the pools, the application can use them as in the following example:

```
ArrayList list = null;
try
{
 list = (ArrayList)arrayListPool.getObject();
 list.clear(); // just in case
 for(int i = 0; i < 10; ++i)
 {
  list.add("" + i);
 }
 // do what ever we need with the ArrayList
}
finally
{
 if(list != null) arrayListPool.returnObject(list);
}
```

This example presents the basic pattern for using object pooling. If the application does not return the object, then the only adverse effect is that the object cannot be reused.

Object pool managers

Object pool managers control the reuse of application objects and Developer Kit objects, such as Vectors and HashMaps.

Multiple object pool managers can be created in an Application Server cell. Each object pool manager has a unique cell-wide Java Naming and Directory Interface (JNDI) name. Applications can find a specific object pool manager by doing a JNDI lookup using the specific JNDI name.

The object pool manager and its associated objects implement the following interfaces:

```
public interface ObjectPoolManager
{
 ObjectPool getPool(Class aClass)
  throws InstantiationException, IllegalAccessException;
 ObjectPool createFastPool(Class aClass)
  throws InstantiationException, IllegalAccessException;

}

public interface ObjectPool
{
 Object getObject();
 void returnObject(Object o);

}
```

The getObject() method removes the object from the object pool. If a getObject() call is made and the pool is empty, then an object of the same type is created. A returnObject( ) call puts the object back into the object pool. If returnObject() is not called, then the object is no longer allocatable from the object pool. If the object is not returned to the object pool, then it can be garbage collected.

Each object pool manager can be used to pool any Java object with the following characteristics:
- The object must be a public class with a public default constructor.
- If the object implements the java.util.Collection interface, it must support the optional clear() method.

Each pooled object class must have its own object pool. In addition, an application gets an object pool for a specific object using either the ObjectPoolManager.getPool() method or the ObjectPoolManager.createFastPool() method. The difference between these methods is that the getPool() method returns a pool that can be shared across multiple threads. The createFastPool() method returns a pool that can only be used by a single thread.

If in a Java virtual machine (JVM), the getPool() method is called multiple times for a single class, the same pool is returned. A new pool is returned for each call when the createFastPool() method is called. Basically, the getPool() method returns a pool that is thread-synchronized.

The pool for use by multiple threads is slightly slower than a fast pool because of the need to handle thread synchronization. However, extreme care must be taken when using a fast pool.

Consider the following interface:

```
public interface PoolableObject
{
 void init();
 void returned();
}
```

If the objects placed in the pool implement this interface and the ObjectPool.getObject() method is called, the object that the pool distributes has the init() method called on it. When the ObjectPool.returnObject() method is called, the PoolableObject.returned() method is called on the object before it is returned to the object pool. Using this method objects can be pre-initialized or cleaned up.

It is not always possible for an object to implement PoolableObject. For example, an application might want to pool ArrayList objects. The ArrayList object needs clearing each time the application reuses it. The application might extend the ArrayList object and have the ArrayList object implement a poolable object. For example, consider the following:

```
public class PooledArrayList extends ArrayList implements PoolableObject
{
 public PooledArrayList()
 {
 }

 public void init() {
 }

 public void returned()
 {
  clear();
 }
}
```

If the application uses this object, in place of a true ArrayList object, the ArrayList object is cleared automatically when it is returned to the pool.

Clearing an ArrayList object simply marks it as empty and the array backing the ArrayList object is not freed. Therefore, as the application reuses the ArrayList, the backing array expands until it is big enough for all of the application requirements. When this point is reached, the application stops allocating and copying new backing arrays and achieves the best performance.

It might not be possible or desirable to use the previous procedure. An alternative is to implement a custom object pool and register this pool with the object pool manager as the pool to use for classes of

that type. The class is registered by the WebSphere administrator when the object pool manager is defined in the cell. Take care that these classes are packaged in Java Archive (JAR) files available on all of the nodes in the cell where they might be used.

Object pool managers collection

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers**.

To use an object pool, the product administrator must define an object pool manager using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Name

Specifies the name by which the object pool manager is known for administrative purposes.

Information	Value
Data type	String
Range	1 through 30 ASCII characters

JNDI name

Specifies the Java Naming and Directory Interface (JNDI) name for the object pool manager.

Information	Value
Data type	String

Scope

Specifies the scope of the configured resource. This value indicates the location for the configuration file.

Description

Specifies the description of the object pool manager.

Information	Value
Data type	String

Category

Specifies the category name used to classify or group this object pool manager.

Information	Value
Data type	String

Object pool managers settings

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers >** *objectpoolmanager_name*

To use an object pool, the product administrator must define an object pool manager using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Scope:

Specifies the scope of the configured resource. This value indicates the location for the configuration file.

Name:

The name by which the object pool manager is known for administrative purposes.

Information	Value
Data type	String
Range	1 through 30 ASCII characters

JNDI Name:

The Java Naming and Directory Interface (JNDI) name for the object pool manager.

Information	Value
Data type	String

Description:

A description of the object pool manager.

Information	Value
Data type	String

Category:

A category name used to classify or to group this object pool manager.

Information	Value
Data type	String

Custom object pool collection:

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers >** *objectpoolmanager_name* **> Custom object pools**.

Use custom object pools to insert additional logic around the following mechanisms:

- Constructing an object pool (A list of properties can be set)
- Flushing the object pool

- Getting objects from the pool
- Returning objects from the pool

These features allow for actions such as, clearing the state of an object when returning it to the pool, configuring the state of an object when retrieving it from the pool, or configuring generic pools and sending instructions on how to behave using custom properties.

To use an object pool the product administrator must define an object pool manager using the administrative console. You can create multiple object pool managers in an Application Server cell.

Pool class name:

Specifies the fully qualified class name of the objects that are stored in the custom object pool.

Information	**Value**
Data type	String

Pool implementation class name:

Specifies the fully qualified class name of the implementation class for the custom object pool.

Information	**Value**
Data type	String

Custom object pool settings:

An object pool manages a pool of arbitrary objects and helps applications avoid creating new Java objects repeatedly. Most objects can be created once, used and then reused. An object pool supports the pooling of objects waiting to be reused. These object pools are not meant to be used for pooling Java Database Connectivity connections or Java Message Service (JMS) connections and sessions. WebSphere Application Server provides specialized mechanisms for dealing with those types of objects. These object pools are intended for pooling application-defined objects or basic Developer Kit types.

To view this administrative console page, click **Resources > Object pool managers >** *objectpoolmanager_name* **> Custom object pools >** *objectpool_name*.

Use custom object pools to insert additional logic around the following mechanisms:

- Constructing an object pool (A list of properties can be set)
- Flushing the object pool
- Getting objects from the pool
- Returning objects from the pool

These features allow for actions such as, clearing the state of an object when returning it to the pool, configuring the state of an object when retrieving it from the pool, or configuring generic pools and sending instructions on how to behave using custom properties.

To use an object pool, the product administrator must define an object pool manager using the administrative console. Multiple object pool managers can be created in an Application Server cell.

Pool Class Name:

The fully qualified class name of the objects that are stored in the object pool.

Information	**Value**
Data type	String

Pool Impl Class Name:

The fully qualified class name of the CustomObjectPool implementation class for this object pool.

Information	Value
Data type	String

Object pool service settings

Use this page to enable or disable the object pool service, which manages object pool resources used by the server.

To view this administrative console page, click **Servers > Server Types > WebSphere application servers >** *server_name* > Container Services > **Object pool service**.

Enable service at server startup

Specifies whether the server attempts to start the object pool service.

Information	Value
Default	Cleared
Range	**Selected** When the application server starts, it attempts to start the object pool service automatically. **Cleared** The server does not try to start the object pool service. If object pool resources are used on this server, then the system administrator must start the object pool service manually or select this property, and then restart the server.

Object pools: Resources for learning

This topic provides links to find relevant supplemental information about object pools.

Use the following links to find relevant supplemental information about object pools. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Furthermore, these links provide guidance on using object pools. Since object pooling is a general topic and the WebSphere Application Server product implementation is only one way to use it, you must understand when object pooling is necessary. These articles help you make that decision.

Programming model and decisions

- Build your own ObjectPool in Java to boost application speed
- Improve the robustness and performance of your ObjectPool
- Recycle broken objects in resource pools

MBeans for object pool managers and object pools

Legacy MBean names for object pool managers and object pools are deprecated. The legacy names are based on the object pool manager name (which is not required to be unique) rather than the object pool manager JNDI name.

About this task

For object pools, the legacy name is also lacking any identifier of the version of the pooled class. Additionally, object pool Performance Monitoring Instrumentation (PMI) statistics are aggregated for object pools with the same legacy object pool MBean name.

For example, if the object pool manager and pooled class are as follows:

```
object pool manager name:               My ObjectPool
object pool manager JNDI name:          op/MyObjectPool
pooled class name:                      java.util.ArrayList
hash code of java.util.ArrayList.class: 1111eb3f (hexadecimal)
```

the legacy object pool manager MBean name will be:

```
ObjectPoolManager_My ObjectPool
```

and the legacy object pool MBean name will be:

```
ObjectPool_My ObjectPool_java.util.ArrayList
```

Instead of using the deprecated legacy MBean names, use the MBean names that are based on the JNDI name of the object pool manager.

For the example above, the JNDI name-based object pool manager MBean name is:

```
ObjectPoolManager_op/MyObjectPool
```

and the JNDI name-based object pool MBean name is:

```
ObjectPool_op/MyObjectPool_java.util.ArrayList.class@1111eb3f
```

Formats for MBean names

Type	Name format
Deprecated legacy object pool manager MBean name:	ObjectPoolManager_[object pool manager name]
JNDI name-based object pool manager MBean name:	ObjectPoolManager_[object pool manager JNDI name]
Deprecated legacy object pool MBean name:	ObjectPool_[object pool manager name]_[pooled class name]
JNDI name-based object pool MBean name:	ObjectPool_[object pool manager JNDI name]_[pooled class name].class@[hexadecimal representation of the hash code of the pooled class' java.lang.Class reference]

In all of the above formats, characters that are not valid for MBean names are replaced with the '.' character.

Chapter 16. Developing Object Request Broker (ORB)

This page provides a starting point for finding information about the Object Request Broker (ORB). The product uses an ORB to manage communication between client applications and server applications as well as among product components. These Java Platform, Enterprise Edition (Java EE) standard services are relevant to the ORB: Remote Method Invocation/Internet Inter-ORB Protocol (RMI/IIOP) and Java Interface Definition Language (Java IDL).

The ORB provides a framework for clients to locate objects in the network and call operations on those objects as though the remote objects were located in the same running process as the client, providing location transparency.

Developing Object Request Brokers

Client-side programming tips for the Object Request Broker service

Every Internet InterORB Protocol (IIOP) request and response exchange consists of a client-side ORB and a server-side ORB. It is important that any application that uses IIOP is properly programmed to communicate with the client-side Object Request Broker (ORB).

The following tips should help you ensure that an application that uses IIOP to handle request and response exchanges is properly programmed to communicate with the client-side Object Request Broker (ORB).

Resolution of initial references to services

Client applications can use the ORBInitRef and ORBDefaultInitRef properties to configure the network location that the ORB service uses to find a service such as naming. When set, these properties are included in the parameters that are used to initialize the ORB, as illustrated in the following example:

```
org.omg.CORBA.ORB.init(java.lang.String[] args,
                       java.util.Properties props)
```

You can set these properties in client code or by command-line argument. It is possible to specify more than one service location by using multiple ORBInitRef property settings (one for each service), but only a single ORBDefaultInitRef value can be specified.

For setting in client code, these properties are com.ibm.CORBA.ORBInitRef.*service_name* and com.ibm.CORBA.ORBDefaultInitRef, respectively. For example, to specify that the naming service (NameService) is located in sample.server.com at port 2809, set the com.ibm.CORBA.ORBInitRef.NameService property to `corbaloc::sample.server.com:2809/NameService`.

For setting by command-line argument, these properties are -ORBInitRef and -ORBDefaultInitRef, respectively. To locate the same naming service specified previously, use the following Java command:

After these properties are set for services that the ORB supports, Java Platform, Enterprise Edition (Java EE) applications can call the resolve_initial_references function on the ORB, as defined in the CORBA/IIOP specification, to obtain the initial reference to a given service.

Preferred API for obtaining an ORB instance

For Java EE applications, you can use either of the following approaches. However, it is strongly recommended that you use the Java Naming and Directory Interface (JNDI) approach to ensure that the same ORB instance is used throughout the client application; you avoid the unintended inconsistencies that might occur when different ORB instances are used.

JNDI approach: For Java EE applications (including enterprise beans, Java EE clients and servlets), you can obtain an ORB instance by creating a JNDI InitialContext object and looking up the ORB under the `java:comp/ORB` name, as illustrated in the following example:

```
javax.naming.Context ctx = new javax.naming.InitialContext();
org.omg.CORBA.ORB orb =
   (org.omg.CORBA.ORB)javax.rmi.PortableRemoteObject.narrow(ctx.lookup("java:comp/ORB"),
                                                            org.omg.CORBA.ORB.class);
```

The ORB instance obtained using JNDI is a singleton object, shared by all the Java EE components that are running in the same Java virtual machine process.

Note: You must use the JNDI approach if you want to take advantage of WLM functionality and cluster failover within the application. For information on how to obtain an InitialContext from a server cluster, see the example for using a CORBA object URL with multiple name server addresses, which is in the topic on getting an initial context by setting the provider URL property.

CORBA approach: Because thin-client applications do not run in a Java EE container, they cannot use JNDI interfaces to look up the ORB. In this case, you can obtain an ORB instance by using CORBA programming interfaces, as follows:

```
java.util.Properties props = new java.util.Properties();
java.lang.String[] args = new java.lang.String[0];
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
```

In contrast to the JNDI approach, the CORBA specification requires that a new ORB instance be created each time the ORB.init method is called. If necessary to change the ORB default settings, you can add ORB property settings to the Properties object that is passed in the ORB.init method call.

The use of the com.ibm.ejs.oa.EJSORB.getORBinstance method, supported in previous releases of this product is deprecated.

API restrictions with sharing an ORB instance among Java EE application components

For performance reasons, it often makes sense to share a single ORB instance among components in a Java EE application. As required by the Java EE Specification, Version 1.3, all web and EJB containers provide an ORB instance in the JNDI namespace as `java:comp/ORB`. Each container can share this instance among application components but is not required to. For proper isolation between application components, application code must comply with the following restrictions:

- Do not call the ORB shutdown or destroy methods
- Do not call org.omg.CORBA_2_3.ORB methods register_value_factory, or unregister_value_factory

In addition, do not share an ORB instance among application components in different Java EE applications.

Required use of rmic and idlj that ship with the IBM Developer Kit

The Java Runtime Environment (JRE) used by this product includes the **rmic** and **idlj** tools. You use the tools to generate Java language bindings for the CORBA/IIOP protocol.

During product installation, the tools are installed in the *app_server_root*`/java/ibm_bin` directory. Versions of these tools included with Java development kits in the `$JAVA_HOME/bin` directory other than the IBM Developer Kit installed with this product are incompatible with this product.

When you install this product, the *app_server_root*`/java/ibm_bin` directory is included in the $PATH search order to enable use of the rmic and idlj scripts provided by IBM. Because the scripts are in the *app_server_root*`/java/ibm_bin` directory instead of the JRE standard *app_server_root*`/java/bin` directory, it is unlikely that you can overwrite them when applying maintenance to a JRE not provided by IBM.

In addition to the rmic and idlj tools, the JRE also includes Interface Definition Language (IDL) files. The files are based on those defined by the Object Management Group (OMG) and can be used by applications that need an IDL definition of selected ORB interfaces. The files are placed in the *app_server_root*/java/ibm_lib directory.

Before using either the rmic or idlj tool, ensure that the *app_server_root*/java/ibm_bin directory is included in the proper PATH variable search order in the environment. If your application uses IDL files in the *app_server_root*/java/ibm_lib directory, also ensure that the directory is included in the PATH variable.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

  The corresponding product variable is *smpe.install.root*.

  The default is /usr/lpp/zWebSphere/V8R5.

Chapter 17. Developing OSGi applications

You can develop bundles, optionally group them into composite bundles, and add them to an OSGi application or a bundle repository.

Procedure

- Develop an OSGi application.

  As an introduction to developing an OSGi application, you can develop a simple hello-world OSGi application, which consists of two bundles. One bundle defines a hello service, and the other is a client bundle that uses this service to produce the message “OSGi Service: Hello World!”.
- Develop a composite bundle.

  A composite bundle groups shared bundles together into aggregates. It provides one or more packages at specific versions to an OSGi application. You can also extend a deployed application by adding one or more composite bundles to the composition unit for the application.
- Convert existing applications to OSGi applications.

  You can convert an enterprise application or a Spring application to an OSGi application.

What to do next

You might also want to explore the sample OSGi applications.

OSGi application design guidelines

When developing OSGi applications for WebSphere Application Server, consider these design guidelines to make the most efficient use of OSGi technology.

The design guidelines described in this topic are:

1. Use OSGi services to configure EJB dependencies
2. Do not use the java:global or java:app namespaces

Details for each of these guidelines are provided in the sections that follow.

1. Use OSGi services to configure EJB dependencies

If your OSGi application has a client bundle that references an EJB in a service bundle, use OSGi services to configure the EJB dependency.

Complete the following steps:

1. Declare the EJB in the Export-EJB header in the bundle manifest file of the service bundle, so that the EJB is registered in the OSGi service registry.
2. Use a reference element in the Blueprint XML file of the client bundle to inject and call the EJB; for more information, see References and the Blueprint Container

Here’s why

Configuring EJB dependencies by using OSGi services reduces the risk of the web or EJB container of the client bundle being recycled, or the client bundle itself being restarted, either of which might result in the temporary unavailability of one or more application endpoints.

2. Do not use the java:global or java:app namespaces

Do not use the `java:global` or `java:app` namespace references to bind to EJBs unless necessary.

Here's why

These namespace references have the following fixed formats:

```
java:global/application_name/module_name/ejb_name
java:app/module_name/ejb_name
```

However, the constituent parts of these references can change after an OSGi application is updated, so that the references must be modified accordingly, which makes this approach impractical. Furthermore, the exposure of the name of the underlying EJB in the reference, rather than using an abstraction such as an intermediate JNDI name, violates the principles of modular application design.

Instead, use OSGi services to configure EJB dependencies, or declare an EJB reference and map it to the EJB JNDI name.

Note: If you define `java:global` or `java:app` references in an @EJB annotation, or in a binding file, you receive a warning when you deploy the OSGi application as a business-level application.

Developing an OSGi application

As an introduction to developing an OSGi application, you can develop a simple hello-world OSGi application, which consists of two bundles. One bundle defines a hello service, and the other is a client bundle that uses this service to produce the message "OSGi Service: Hello World!".

About this task

An OSGi application is a Java application that uses OSGi technologies. OSGi applications are collections of OSGi bundles (typically bundles that use the Blueprint component model), and can expose or consume a number of services. The OSGi application described in these topics demonstrates the use of the OSGi service registry to share the hello service between the defining bundle and the client bundle. All interactions with the service registry are handled through Blueprint.

OSGi bundles are packaged as Java archive (.jar) files. A single OSGi application is packaged in an enterprise bundle archive (.eba) file, just as an enterprise application is packaged in an enterprise archive (.ear) file. In this example application, the bundles are packaged directly in the .eba file. However the .eba file does not have to contain the bundles; they can be pulled in at run time.

Note: The steps in the following procedure are specific to this example application, and lead you through creating the application artifacts by using IBM Rational Application Developer Version 8 or a similar tool.

Procedure

1. Create your service bundle.
2. Create your client bundle.
3. Create your OSGi application.

What to do next

After you create your OSGi application, you can deploy the application to WebSphere Application Server by using either the administrative console or wsadmin commands; for details, see "Deploying an OSGi application as a business-level application" on page 2091.

You might also want to explore the sample OSGi applications.

Creating a service bundle

For the simple hello-world OSGi application, the service bundle implements the HelloWorldEBA interface, and exports it as an OSGi service.

About this task

A bundle, the modular unit in the OSGi model, is a JAR file that includes the OSGi application metadata. This metadata is defined in the manifest file of the JAR file, `META-INF/MANIFEST.MF`.

IBM Rational Application Developer Version 8 provides graphical support for creating and packaging bundles. The sample procedure that follows uses this tool. You can also use other tools, and the steps are adaptable to other tools.

This sample procedure builds the following two bundles.

- `com.ibm.ws.eba.helloWorld.api`: this bundle declares the HelloWorldEBA interface.
- `com.ibm.ws.eba.helloWorld.service`: this bundle implements the HelloWorldEBA interface, and exports it as an OSGi service. The exported service is used by client bundle `com.ibm.ws.eba.helloWorld.client`, as described in "Creating a client bundle" on page 665.

Procedure

1. Create the com.ibm.ws.eba.helloWorld.api bundle. This bundle declares the HelloWorldEBA interface.
   a. Click **File** > **New** > **OSGi Bundle Project**. The OSGi Bundle Project panel is displayed.
   b. Configure the project.
      - For the **Project name**, enter `com.ibm.ws.eba.helloWorld.api`.
      - Clear the **Add bundle to application** check box. If you leave this check box selected, a new OSGi application project is created automatically, and the bundle is added to it. Here, however, the application project will be created manually in a separate task, "Creating an OSGi application" on page 668.
      - Leave the other options as the default values.
   c. Click **Next**. The Java Configuration panel is displayed. Accept the default values for all the options on this panel.
   d. Click **Next**. The OSGi bundle settings panel is displayed. Accept the default values for all the options on this panel.
   e. Click **Finish**.

   You have created your OSGi project.
2. Declare the `HelloWorldEBA` interface.

   Create a package called `com.ibm.ws.eba.helloWorld.api`, that includes an interface called `HelloWorldEBA`. Code this interface to contain just one method: `hello()`.
   a. Under your `com.ibm.ws.eba.helloWorld.api` project, right-click the folder `src`, then select **New** > **Package**.
   b. Name the new package `com.ibm.ws.eba.helloWorld.api`.
   c. Click **Finish**.
   d. Right-click the new package, then select **New** > **Interface**.
   e. Name the new interface `HelloWorldEBA`.
   f. Click **Finish**.
   g. Copy and paste the following method to replace the content of the interface file:

```
package com.ibm.ws.eba.helloWorld.api;
public interface HelloWorldEBA {

    public void hello();
}
```

h. Save and close the file.

3. Configure the com.ibm.ws.eba.helloWorld.api package as an exported package.

   Edit the bundle manifest in the com.ibm.ws.eba.helloWorld.api project to allow other bundles to load classes from the com.ibm.ws.eba.helloWorld.api package. Classes that are in packages not exported in the bundle manifest are private to the defining bundle and cannot be loaded by any other bundle.

   a. Open the bundle MANIFEST.MF file with the manifest editor. This file is in the BundleContent/META-INF directory.
   b. Click the **Runtime** tab.
   c. In the Exported Packages pane, click **Add**.
   d. Select the com.ibm.ws.eba.helloWorld.api package from the list, then click **OK**.
   e. In the Exported Packages pane, click **Properties**.
   f. In the **Properties** dialog, set the **version** to 1.0.0, then click **OK**.
   g. Save and close the file.

4. Create the com.ibm.ws.eba.helloWorld.service bundle. This bundle implements the HelloWorldEBA interface.

   a. Click **File** > **New** > **OSGi Bundle Project**. The OSGi Bundle Project panel is displayed.
   b. Configure the project.
      - For the **Project name**, enter com.ibm.ws.eba.helloWorld.service.
      - Clear the **Add bundle to application** check box. If you leave this check box selected, a new OSGi application project is created automatically, and the bundle is added to it. Here, however, the application project will be created manually in a separate task, “Creating an OSGi application” on page 668.
      - Leave the other options as the default values.
   c. Click **Next**. The Java Configuration panel is displayed. Accept the default values for all the options on this panel.
   d. Click **Next**. The OSGi bundle settings panel is displayed. Accept the default values for all the options on this panel.
   e. Click **Finish**.

5. Make the HelloWorldEBA interface available to the service implementation bundle.

   Edit the client bundle manifest to make classes inside the com.ibm.ws.eba.helloWorld.api package available to the service implementation bundle. The com.ibm.ws.eba.helloWorld.api package is part of the com.ibm.ws.eba.helloWorld.api bundle.

   a. Expand the com.ibm.ws.eba.helloWorld.service project.
   b. Open the bundle MANIFEST.MF file with the manifest editor. This file is in the BundleContent/META-INF directory.
   c. Click the **Dependencies** tab.
   d. In the Imported Packages pane, click **Add**.
   e. In the Package Selection dialog, enter com.ibm.ws.eba, select com.ibm.ws.eba.helloWorld.api from the Exported Packages list, then click **OK**. The package is added to the Imported Packages list.
   f. In the Imported Packages list, select the com.ibm.ws.eba.helloWorld.api package then click **Properties**.
   g. In the **Properties** dialog, set the **minimum version** to 1.0.0 Inclusive, and set the **maximum version** to 1.1.0 Exclusive, then click **OK**. The entry for this package in the Imported Packages list is updated to com.ibm.ws.eba.helloWorld.api [1.0.0,1.1.0).

This version syntax means “exported packages with versions between 1.0.0 inclusive and 1.1.0 exclusive will match this import”. For more information on the version syntax, see section 3.2.6 “Version Ranges” of the OSGi Service Platform Release 4 Version 4.2 Core Specification.

This version range has been specified to ensure that the implementation bundle uses an updated version of the package only if it differs in the value of the micro version, because a major change, such as removing a method from an interface, or a minor change, such as adding a method to an interface, could cause the implementation bundle to cease functioning correctly.

h. Save and close the file.

6. Implement the HelloWorld service.

   Create a package called `com.ibm.ws.eba.helloWorld.service`, that includes an implementation class called `HelloWorldService`. Code this class to implement the hello() method from the `HelloWorldEBA` interface. This implementation of the class causes “OSGi Service: Hello World!” to be displayed.

   a. Under your `com.ibm.ws.eba.helloWorld.service` project, right-click the folder `src`, then select **New** > **Package**.
   b. Name the new package `com.ibm.ws.eba.helloWorld.service`.
   c. Click **Finish**.
   d. Right-click the new package, then select **New** > **Class**.
   e. Name the new interface implementation class `HelloWorldService`.
   f. Click **Add** alongside the **Interfaces** field.
   g. Enter `Hello`, select `HelloWorldEBA` from the **Matching items** list, then click **OK**.
   h. Ensure that the **Inherited abstract methods** check box is selected.
   i. Click **Finish**.
   j. Provide implementation code for the inherited `hello()` method. Replace the line

      ```
      // TODO Auto-generated method stub
      ```

      with the line

      ```
      System.out.println("OSGi Service: Hello World!");
      ```

      The complete code for the implementation class should now be as follows:

      ```
      package com.ibm.ws.eba.helloWorld.service;

      import com.ibm.ws.eba.helloWorld.api.HelloWorldEBA;

      public class HelloWorldService implements HelloWorldEBA {

          @Override
          public void hello() {
              System.out.println("OSGi Service: Hello World!");
          }

      }
      ```

   k. Save and close the file.

7. Export the helloWorld service by using OSGi Blueprint XML.

   A Blueprint configuration contains the bundle component assembly and configuration information. It also describes how components are registered in the OSGi service registry, or how they look up services from the OSGi service registry. This information is used at run time to instantiate and configure the required components when the bundle is started.

   a. In the project `com.ibm.ws.eba.helloWorld.service`, create a Blueprint XML file:
      1) Right-click the `com.ibm.ws.eba.helloWorld.service` project, and select **New** > **Blueprint File**.
      2) Accept the default values for all the options on this panel.
      3) Click **Finish**.
   b. Add a bean element to the Blueprint XML file.

1) In the **Design** tab, click **Add** in the Overview pane.
2) Select **Bean**, and click **OK**.
3) Click **Browse**, select `HelloWorldService`, and click **OK**.
4) In the **Bean ID** field, enter `HelloEBA`, then click **OK** to add the bean element.

c. Add a service element to the Blueprint XML file.
1) Select **Blueprint** in the Overview pane, and click **Add**.
2) Select **Service**, and click **OK**.
3) Click **Browse** alongside the **Service Interface** field.
4) Enter `Hello`, select `HelloWorldEBA` from the **Matching items** list, and click **OK**.
5) Click **Browse** alongside the **Bean Reference** field, select `HelloEBA`, then click **OK**.
6) Click **OK** to add the service element.

d. Examine the Blueprint XML source code.

Select the **Source** tab. The source code should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="helloEBA"
    class="com.ibm.ws.eba.helloWorld.service.HelloWorldService"/>
  <service id=HelloEBAService ref="helloEBA"
    interface="com.ibm.ws.eba.helloWorld.api.HelloWorldEBA"/>
</blueprint>
```

In the previous code block:

- The `bean` element defines a Blueprint component to be instantiated. In this example, the `bean` element causes bean `helloEBA` to be instantiated, by calling the constructor for the `com.ibm.ws.eba.helloWorld.service.HelloWorldService` class.
  - The **`id`** attribute identifies the bean. You must specify this attribute if the bean is referenced from elsewhere in the Blueprint information, for example from the `service` element.
  - The **`class`** attribute specifies which implementation class of the bean is instantiated.
- The `service` element defines the registration of a component in the OSGi service registry. In this example, the `service` element registers the bean with the name `helloEBA` as a service in the OSGi service registry with interface `com.ibm.ws.eba.helloWorld.api.HelloWorldEBA`, specified by the interface attribute.
  - The **`ref`** attribute refers to the **`id`** of the bean to be registered. This **`id`** is defined in the `bean` element.
  - The **`interface`** attribute refers to the interface that the bean class implements.

For more information, see section 121.5 “Bean Manager” and section 121.6 “Service Manager” of the OSGi Service Platform Release 4 Version 4.2 Enterprise Specification.

e. Save and close the file.

Note: You might get an exception message (visible in the Problems pane) saying that there is no `bin.include` entry for `OSGI-INF` in the build properties file. If you see this message, use the **quick-fix** option to add the entry (right-click the problem, then select **quick-fix**).

Results

You have created two bundles, `com.ibm.ws.eba.helloWorld.api` and `com.ibm.ws.eba.helloWorld.service`. The `com.ibm.ws.eba.helloWorld.service` service implements the HelloWorldEBA interface that is declared in the `com.ibm.ws.eba.helloWorld.api` bundle, and contains the business logic and metadata needed to export the `com.ibm.ws.eba.helloWorld.service` service.

What to do next

You can now create the client bundle that uses the `com.ibm.ws.eba.helloWorld.service` service.

Creating a client bundle

For the simple hello-world OSGi application, the client bundle consumes the HelloWorldService service, and uses it to produce the message “OSGi Service: Hello World!”.

Before you begin

This task assumes that you have already created the HelloWorldService service, as described in “Creating a service bundle” on page 661.

About this task

A bundle, the modular unit in the OSGi model, is a JAR file that includes the OSGi application metadata. This metadata is defined in the manifest file of the JAR file, `META-INF/MANIFEST.MF`.

IBM Rational Application Developer Version 8 provides graphical support for creating and packaging bundles. The sample procedure that follows uses this tool. You can also use other tools, and the steps are adaptable to other tools.

This sample procedure builds a bundle called `com.ibm.ws.eba.helloWorld.client`. This bundle uses the HelloWorldService service that is exported by the bundle `com.ibm.ws.eba.helloWorld.service`, as described in “Creating a service bundle” on page 661.

Procedure

1. Create an OSGi bundle project.
   a. Click **File** > **New** > **OSGi Bundle Project**. The OSGi Bundle Project panel is displayed.
   b. Configure the project.
      - For **Project name**, enter `com.ibm.ws.eba.helloWorld.client`.
      - Clear the **Add bundle to application** check box. If you leave this check box selected, a new OSGi application project is created automatically, and the bundle is added to it. Here, however, the application project will be created manually in a separate task, “Creating an OSGi application” on page 668.
      - Leave the other options as the default values.
   c. Click **Next**. The Java Configuration panel is displayed. Accept the default values for all the options on this panel.
   d. Click **Next**. The OSGi bundle settings panel is displayed. Accept the default values for all the options on this panel.
   e. Click **Finish**.

   You have created your OSGi project.
2. Make the HelloWorldEBA interface available to the client implementation bundle.

   Edit the client bundle manifest to make classes inside the `com.ibm.ws.eba.helloWorld.api` package available to the client implementation bundle. The `com.ibm.ws.eba.helloWorld.api` package is part of the `com.ibm.ws.eba.helloWorld.api` bundle.
   a. Expand the `com.ibm.ws.eba.helloWorld.client` project
   b. Open the bundle `MANIFEST.MF` file with the manifest editor. This file is in the `BundleContent/META-INF` directory.
   c. Click the **Dependencies** tab.
   d. In the Imported Packages pane, click **Add**.

e. In the Package Selection dialog, enter com.ibm.ws.eba, select com.ibm.ws.eba.helloWorld.api (1.0.0) from the Exported Packages list, then click **OK**. The package is added to the Imported Packages list.

f. In the Imported Packages list, select the com.ibm.ws.eba.helloWorld.api package then click **Properties**.

g. In the **Properties** dialog, set the **minimum version** to 1.0.0 Inclusive, and set the **maximum version** to 2.0.0 Exclusive, then click **OK**. The entry for this package in the Imported Packages list is updated to com.ibm.ws.eba.helloWorld.api [1.0.0,2.0.0).

This version syntax means "exported packages with versions between 1.0.0 inclusive and 2.0.0 exclusive will match this import". For more information on the version syntax, see section 3.2.6 "Version Ranges" of the OSGi Service Platform Release 4 Version 4.2 Core Specification.

This version range has been specified to ensure that the client bundle uses an updated version of the package if it differs in the value of the minor version or the micro version, but not the major version, because only a binary incompatible change, such as the deletion of a public method, can cause the client bundle to cease functioning correctly.

h. Save and close the file.

3. Create a class HelloWorldClient.

a. In the project source folder src, create a package called com.ibm.ws.eba.helloWorld.client. In the package, create a class called HelloWorldClient with the following contents:

```
package com.ibm.ws.eba.helloWorld.client;

import com.ibm.ws.eba.helloWorld.api.HelloWorldEBA;

public class HelloWorldClient {
    private HelloWorldEBA helloWorldEBAService = null; //a reference to the service

    public void refHello() {
        System.out.println("Client: Start...");
        helloWorldEBAService.hello();
        System.out.println("Client: End...");
    }

    public HelloWorldEBA getHelloWorldEBAService() {
        return helloWorldEBAService;
    }

    public void setHelloWorldEBAService(HelloWorldEBA helloWorldEBAService) {
        this.helloWorldEBAService = helloWorldEBAService;
    }
}
```

In the previous code block:

- The line HelloWorldEBA helloWorldEBAService = null; defines the service dependency.
- The line helloWorldEBAService.hello(); demonstrates that a service has been injected for the helloWorldEBAService dependency.

b. Save and close the file.

4. Create a Blueprint configuration.

A Blueprint configuration contains the bundle component assembly and configuration information. It also describes how components are registered in the OSGi service registry, or how they look up services from the OSGi service registry. This information is used at run time to instantiate and configure the required components when the bundle is started.

a. In the project com.ibm.ws.eba.helloWorld.client, create a Blueprint XML file:

1) Right-click the com.ibm.ws.eba.helloWorld.client project, and select **New** > **Blueprint File**.

2) (Optional) Specify the file name. This can be any name provided it is an XML file. For example, helloWorldRef.xml.

3) Leave the other options as the default values.

4) Click **Finish**.

b. Add a reference element to the Blueprint XML file.
   1) In the **Design** tab, click **Add** in the Overview pane.
   2) Select **Reference**, and click **OK**.
   3) Click **Browse** alongside the **Reference Interface** field.
   4) Enter `Hello`, select `HelloWorldEBA` from the **Matching items** list, and click **OK**.
   5) In the **Reference ID** field, enter `helloEBARef`, then click **OK** to add the reference element.

c. Add a bean element to the Blueprint XML file.
   1) Select **Blueprint** in the Overview pane, and click **Add**.
   2) Select **Bean**, and click **OK**.
   3) Click **Browse**, select `HelloWorldClient`, and click **OK**.
   4) Click **OK** to add the bean element.
   5) In the Method References pane, click **Browse** alongside the **Initialization method** field.
   6) Select **refHello()** and click **OK**

d. Add a property to the bean.
   1) Select **HelloWorldClientBean** in the Overview pane, and click **Add**.
   2) Select **Property**, and click **OK**.
   3) In the Details pane, enter `helloWorldEBAService` in the **Name** field.
   4) Click **Browse** alongside the **Reference** field.
   5) Select **helloEBARef** and click **OK**.

e. Examine the Blueprint XML source code. This Blueprint file specifies the internal wiring of the components.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <reference id="helloEBARef"
    interface="com.ibm.ws.eba.helloWorld.api.HelloWorldEBA"/>
  <bean id="HelloWorldClientBean"
    class="com.ibm.ws.eba.helloWorld.client.HelloWorldClient"
    init-method="refHello">
    <property name="helloWorldEBAService" ref="helloEBARef"/>
  </bean>
</blueprint>
```

   In the previous code block:

   - The `init-method refHello` is called when the `com.ibm.ws.eba.helloWorld.client.HelloWorldClient` bean is created.
   - The `property` element specifies how a dependency is injected.
     - The property `helloWorldEBAService` is set by invoking the property setter `public void setHelloWorldEBAService(HelloWorldEBA helloWorldEBAService)`. You can use setter injection for bean properties that are defined in accordance with Java bean conventions.
     - The attribute `ref` specifies the Blueprint element ID of the component to be injected. The `ref` attribute must match a top-level element in this file. In this case, it matches the `reference` element.
   - The `reference` element defines a dependency of this Blueprint module on an OSGi service with interface `com.ibm.ws.eba.helloWorld.api.HelloWorldEBA`. When the Blueprint Container starts this module, it must match the name of the interface attribute with an available service in the OSGi service registry. If this cannot be done, the Blueprint module is not started. In this case, the service is defined in the service element of the Blueprint XML of the service bundle.

   For more information, see section 121.5 “Bean Manager” and section 121.7 “Service Reference Managers” of the OSGi Service Platform Release 4 Version 4.2 Enterprise Specification.

f. Save and close the file.

Results

You have created a bundle called com.ibm.ws.eba.helloWorld.client. This bundle consumes the HelloWorldService service, and uses it to produce the message “OSGi Service: Hello World!”.

What to do next

You can now create an OSGi application, in which multiple bundles are packaged together.

Creating an OSGi application

For the simple hello-world OSGi application, the bundle that defines the service, and the bundle that uses the service, are packaged together in an OSGi application.

Before you begin

This topic assumes that you have already completed the following tasks:

- “Creating a service bundle” on page 661 com.ibm.ws.eba.helloWorld.service.
- “Creating a client bundle” on page 665 com.ibm.ws.eba.helloWorld.client.

You can create your application as described in this topic, in which case the application manifest file is created for you by the tooling, or you can create your application using Apache Ant.

About this task

OSGi bundles are packaged as Java archive (.jar) files. A single OSGi application is packaged in an enterprise bundle archive (.eba) file, just as an enterprise application is packaged in an enterprise archive (.ear) file. In this example application, the bundles are packaged directly in the .eba file. However the .eba file does not have to contain the bundles; they can be pulled in at run time. For an OSGi application deployed to WebSphere Application Server, a bundle can be located in the .eba file, or in the WebSphere Application Server internal bundle repository, or in an external bundle repository. Every .eba file contains an application manifest (APPLICATION.MF) file, which contains the metadata that defines the application. It lists the bundles that the application uses, and says where each bundle is located.

IBM Rational Application Developer Version 8 provides graphical support for creating and packaging bundles. The sample procedure that follows uses this tool. You can also use other tools, and the steps are adaptable to other tools.

This sample procedure creates a simple hello-world OSGi application called com.ibm.ws.eba.helloWorldApp, in which the three bundles com.ibm.ws.eba.helloWorld.api, com.ibm.ws.eba.helloWorld.service and com.ibm.ws.eba.helloWorld.client are packaged together in an .eba file.

When you deploy and start this application, it prints the greeting message “OSGi Service: Hello World!” to the system output log.

Procedure

1. Create an OSGi application project.
   a. Click **File** > **New** > **OSGi Application Project**. The OSGi Application Project panel is displayed.
   b. Configure the project.
      - For **Project name**, enter com.ibm.ws.eba.helloWorldApp.
      - Ensure that **Use default location** is selected.
      - Ensure that **Add project to working sets** is cleared.
   c. Click **Next**.

d. In the Contained Bundles list, select `com.ibm.ws.eba.helloWorld.api 1.0.0`, `com.ibm.ws.eba.helloWorld.client1.0.0` and `com.ibm.ws.eba.helloWorld.service1.0.0`.

   For more information about these bundles, see “Creating a service bundle” on page 661 and “Creating a client bundle” on page 665.

e. Click **Finish**.

2. Optional: Deploy and test the OSGi application project on a server running inside IBM Rational Application Developer.

   If you are using IBM Rational Application Developer Version 8, you can deploy to a local machine without first exporting the OSGi application as an EBA file. This is typically how you would test your application. If you are using another tool, omit this step.

   a. Right-click the `com.ibm.ws.eba.helloWorldApp` project, then select **Run As** > **Run on Server**. The Run On Server panel is displayed. If you have already configured a WebSphere Application Server in your IBM Rational Application Developer workspace, you can select **Choose an existing server**, and then select that server, otherwise manually define a new server by completing the remaining steps.

   b. Configure the Run On Server panel:

      1) For server type, select **IBM** > **WebSphere Application Server v8.0**.
      2) For server host name, enter a valid TCP/IP host name. For example, enter the TCP/IP hostname for your computer or enter `localhost`.
      3) For server name, enter the name by which you want to refer to the server within IBM Rational Application Developer.
      4) For server runtime environment, select **WebSphere Application Server v8.0**. If this option is not available in the list, complete the following steps:
         a) Click **Add**.
         b) Navigate to the *app_server_root* directory for your installation of WebSphere Application Server; for more information on the *app_server_root* directory, see *app_server_root*.
         c) Click **Finish**.
      5) Click **Next**.

      The WebSphere Application Server Settings panel is displayed.

   c. Configure the WebSphere Application Server Settings panel:

      1) For the profile name, select the profile that you use with OSGi Applications.
      2) For the application server name, verify that the application server name for your profile is displayed. By default, the first application server for a profile is called `server1`.
      3) If security is enabled on your application server, enter a user ID and password.
      4) Click **Next**.

      The list of configured OSGi application projects is displayed.

   d. Check that your OSGi application project is in the list of configured projects.

   e. Click **Finish**. The server starts automatically, your application is published and started, and the following output is displayed on the **Console** tab:

```
[4/15/10 14:07:33:295 GMT] 00000023 SystemOut O Client: Start...
[4/15/10 14:07:33:581 GMT] 00000023 SystemOut O OSGi Service: Hello World!
[4/15/10 14:07:33:581 GMT] 00000023 SystemOut O Client: End...
```

3. Export the project as an EBA file.

   a. Right-click the project name, then select **Export** > **OSGi** > **OSGi Application EBA**.

   b. Enter a location for the exported file, then name the file `com.ibm.ws.eba.helloWorldApp.eba`.

   c. Click **Finish**.

Results

You have completed the detailed instructions for developing a simple OSGi application. In this application, a client bundle that uses a service that is defined in a service bundle to produce the message “OSGi Service: Hello World!”. Your application uses the OSGi service registry to share the hello service between the defining bundle and the client bundle. All interactions with the service registry are handled through Blueprint.

What to do next

You are now ready to deploy and start your OSGi application outside of IBM Rational Application Developer, by completing the following steps:

1. Deploy your OSGi application as a business-level application.
2. Start your business-level application.
3. Check the system output log for the greeting message “OSGi Service: Hello World!”.

You can use the administrative console or wsadmin commands to deploy and start an OSGi application in WebSphere Application Server. For information about how to do this for any OSGi application, see “Deploying an OSGi application as a business-level application” on page 2091 and Starting your business-level application. When you check the system output log, you should see the following message:

```
[4/15/10 14:07:33:295 GMT] 00000023 SystemOut O Client: Start...
[4/15/10 14:07:33:581 GMT] 00000023 SystemOut O OSGi Service: Hello World!
[4/15/10 14:07:33:581 GMT] 00000023 SystemOut O Client: End...
```

Creating an OSGi application using Apache Ant

You can use the Apache Ant (Ant) command-line tool to package bundles together into an OSGi application.

Before you begin

This topic assumes that you understand how to use the Ant build tooling, and that you have already created the bundles and the application manifest file that you want to package together as an OSGi application.

For a detailed overview of creating an OSGi application, and instructions on how to do this using IBM Rational Application Developer Version 8 or similar tooling, see “Creating an OSGi application” on page 668.

About this task

The code example in this topic shows the syntax for using the Ant **zip** task to package a set of bundles and an application manifest (`APPLICATION.MF`) file into an enterprise bundle archive (`.eba`) file.

Example

```
<zip destfile="${output.dir}/myExample.eba" basedir="${basedir}">
    <filename name="META-INF/APPLICATION.MF"/>
    <fileset dir="${basedir}">
        <include name="*.jar"/>
    </fileset>
</zip>
```

This example packages the `META-INF/APPLICATION.MF`file into the EBA file, and also includes all files ending in “`.jar`” (that is, the bundles).

`${output.dir}` and `${basedir}` refer to the following Ant properties:

- **output.dir** is user-defined, and specifies the output directory for your build.

- **basedir** is predefined, and specifies the directory that contains the Ant `build.xml` file.

What to do next

You can use the administrative console or wsadmin commands to deploy and start an OSGi application in WebSphere Application Server. See “Deploying an OSGi application as a business-level application” on page 2091 and Starting your business-level application.

Developing a composite bundle

A composite bundle groups shared bundles together into aggregates. It provides one or more packages at specific versions to an OSGi application. You can also extend a deployed application by adding one or more composite bundles to the composition unit for the application. You can use OSGi application tooling to develop a composite bundle.

About this task

When you want to ensure consistent behavior from a set of shared bundles in an OSGi application, you use a composite bundle to provide that set of bundles to the application. If a required package or service is available at the same version from both a bundle and a composite bundle, the provisioning process selects the package or service from the composite bundle.

When you want to extend a deployed business-level application that contains an OSGi application, and you do not want to stop the application or modify the underlying EBA asset, you add one or more composite bundles to the composition unit.

A composite bundle is packaged as a composite bundle archive (CBA) file. This file is a compressed archive file with a `.cba` file extension. If the composite bundle is part of an enterprise OSGi application, the CBA file can be directly contained within the enterprise bundle archive (EBA) file for the application, or pulled in by reference from the internal bundle repository or from an external repository that can process composite bundles. A composite bundle can directly contain bundles in its CBA file. It can also include by reference bundles that are hosted alongside the CBA file within the same EBA file, or bundles that are installed in the same bundle repository.

A composite bundle is described in a composite bundle manifest file, `META-INF/COMPOSITEBUNDLE.MF`. This manifest file lists the OSGi bundles that are directly contained in the composite bundle, and the reference bundles that are hosted alongside the composite bundle in the same EBA file, or in the same bundle repository.

If you want to use your composite bundle to extend a deployed application, you must install the CBA file in the internal bundle repository or in an external repository that can process composite bundles. If you install a composite bundle in a bundle repository, and the composite bundle includes bundles by reference, you must ensure that the referenced bundles are also available in the same repository. If you use the internal bundle repository, and the composite bundle directly contains bundles, the contained bundles are not listed separately and are only available as part of the composite bundle. For more information, see Composite bundles.

IBM Rational Application Developer Version 8 provides graphical support for creating and packaging composite bundles. You can also use other tools.

Procedure

1. Develop the bundles.

   See “Creating a service bundle” on page 661 and “Creating a client bundle” on page 665. You might also want to explore the sample OSGi applications.
2. Create the composite bundle manifest.

Most OSGi application tooling, including IBM Rational Application Developer Version 8, helps you create the composite bundle manifest. See also Example: OSGi composite bundle manifest file.

3. Package the directly-contained bundles and the composite bundle manifest as a compressed file with a `.cba` file extension.

   Note: The file extension must be `.cba`, but the composite bundle name need not include "cba".

4. Check that all referenced bundles are available in the EBA file (if the composite bundle is part of an enterprise OSGi application), or in a bundle repository that can process composite bundles (such as the internal bundle repository).

   See Administering bundles in the internal bundle repository or Administering bundles in the internal bundle repository using wsadmin commands.

5. Add the CBA file to the EBA file or to the bundle repository.

What to do next

- If you have configured an enterprise OSGi application to use your composite bundle, you can now import the application as an asset. See "Deploying an OSGi application as a business-level application" on page 2091.
- If you want to use your composite bundle to extend a composition unit, see Adding or removing extensions for an OSGi composition unit.

Converting existing applications to OSGi applications

You can convert an enterprise application or a Spring application to an OSGi application.

Procedure

- Convert an enterprise application to an OSGi application.
- Convert a Spring application to an OSGi application.

Converting an enterprise application to an OSGi application

You convert an enterprise application to an OSGi application by completing manual tasks that convert the enterprise archive (EAR) file to an enterprise bundle archive (EBA) file.

To convert your enterprise application to an OSGi application, you must separately convert each of the components in the EAR file.

Complete the following tasks:

1. Change the file extension from `.ear` to `.eba`.
2. Convert any web application archive (WAR) files to OSGi web application bundles.
3. Convert any enterprise bean (EJB) Java archive (JAR) files to OSGi EJB bundles.
4. Convert any utility JAR files to OSGi bundles.
5. Convert any persistence archive files to OSGi persistence bundles.
6. Convert Java 2 security settings to OSGi.

These tasks are described in detail in the following subtopics:

Converting a web application archive file to an OSGi web application bundle

When converting an enterprise archive (EAR) file to an enterprise bundle archive (EBA) file, you complete manual tasks to convert any web application archive (WAR) files in the EAR file to OSGi web application bundles.

Note: If you do not convert a WAR file manually, it is converted automatically when the containing EBA file is imported as an asset. However, by completing all of the manual steps described in this topic, you can ensure that your web application bundle is configured correctly for your deployment environment.

To convert a WAR file to an OSGi web application bundle, complete the following steps:

1. Change the file extension from `.war` to `.jar`.
2. Define general bundle metadata by adding the following headers to the bundle manifest file, `META-INF/MANIFEST.MF`:

   Bundle-ManifestVersion
   The version of the syntax in which the bundle manifest file is written. For OSGi Service Platform Release 4, set the value to 2.

   Bundle-Name
   A human-readable name for the bundle.

   Bundle-SymbolicName
   A non-localizable name that identifies the bundle uniquely.

   Bundle-Version
   The version of the bundle. For more information, see the description of the Bundle-Version header in the bundle manifest file.

   Import-Package
   The external packages on which the bundle depends. For more information, see the description of the Import-Package header in the bundle manifest file.

   Export-Package
   The packages that are visible outside the bundle. For more information, see the description of the Export-Package header in the bundle manifest file.
3. Define bundle-type-specific metadata by adding the following headers to the bundle manifest file:

   Export-EJB
   The presence of this header identifies the bundle as containing enterprise beans that are to be loaded and run by the EJB container. Optionally, you can specify, as the value of the this header, the list of enterprise beans that you want to export as OSGi services. For more information, see the description of the Export-EJB header in the bundle manifest file.

   Web-ContextPath
   The default context from which the web content is hosted.

   You must set the value of the Web-ContextPath header to the value of the <context-root> element for the corresponding web module in the application.xml file of the enterprise application EAR file.
4. Specify the bundle classpath by adding a **Bundle-Classpath** header to the bundle manifest file.

   You must set the value of the Bundle-Classpath header to a comma separated list of the names of all the JAR files and class subfolders that are contained in the `WEB-INF\lib` folder of the WAR file.

Converting an EJB JAR file to an OSGi EJB bundle

When converting an enterprise archive (EAR) file to an enterprise bundle archive (EBA) file, you complete manual tasks to convert any enterprise bean (EJB) Java archive (JAR) files in the EAR file to OSGi EJB bundles.

To convert an EJB JAR file to an OSGi application bundle, complete the following steps:

1. Define general bundle metadata by adding the following headers to the bundle manifest file, `META-INF/MANIFEST.MF`:

Bundle-ManifestVersion
The version of the syntax in which the bundle manifest file is written. For OSGi Service Platform Release 4, set the value to 2.

Bundle-Name
A human-readable name for the bundle.

Bundle-SymbolicName
A non-localizable name that identifies the bundle uniquely.

Bundle-Version
The version of the bundle. For more information, see the description of the Bundle-Version header in the bundle manifest file.

Import-Package
The external packages on which the bundle depends. For more information, see the description of the Import-Package header in the bundle manifest file.

Export-Package
The packages that are visible outside the bundle. For more information, see the description of the Export-Package header in the bundle manifest file.

2. Define bundle-type-specific metadata by adding the following headers to the bundle manifest file:

Export-EJB
The presence of this header identifies the bundle as containing enterprise beans that are to be loaded and run by the EJB container. Optionally, you can specify, as the value of the this header, the list of enterprise beans that you want to export as OSGi services. For more information, see the description of the Export-EJB header in the bundle manifest file.

Converting a utility JAR file to an OSGi bundle

When converting an enterprise archive (EAR) file to an enterprise bundle archive (EBA) file, you complete manual tasks to convert any utility JAR files in the EAR file to OSGi bundles.

To convert a utility JAR file to an OSGi application bundle, define general bundle metadata by adding the following headers to the bundle manifest file, META-INF/MANIFEST.MF:

Bundle-ManifestVersion
The version of the syntax in which the bundle manifest file is written. For OSGi Service Platform Release 4, set the value to 2.

Bundle-Name
A human-readable name for the bundle.

Bundle-SymbolicName
A non-localizable name that identifies the bundle uniquely.

Bundle-Version
The version of the bundle. For more information, see the description of the Bundle-Version header in the bundle manifest file.

Import-Package
The external packages on which the bundle depends. For more information, see the description of the Import-Package header in the bundle manifest file.

Export-Package
The packages that are visible outside the bundle. For more information, see the description of the Export-Package header in the bundle manifest file.

:

Converting a persistence archive file to an OSGi bundle

When converting an enterprise archive (EAR) file to an enterprise bundle archive (EBA) file, you complete manual tasks to convert any persistence archive files in the EAR file to OSGi bundles.

To convert a persistence archive file to an OSGi application bundle, complete the following steps:

1. Define general bundle metadata by adding the following headers to the bundle manifest file, `META-INF/MANIFEST.MF`:

   Bundle-ManifestVersion
   The version of the syntax in which the bundle manifest file is written. For OSGi Service Platform Release 4, set the value to 2.

   Bundle-Name
   A human-readable name for the bundle.

   Bundle-SymbolicName
   A non-localizable name that identifies the bundle uniquely.

   Bundle-Version
   The version of the bundle. For more information, see the description of the Bundle-Version header in the bundle manifest file.

   Import-Package
   The external packages on which the bundle depends. For more information, see the description of the Import-Package header in the bundle manifest file.

   Export-Package
   The packages that are visible outside the bundle. For more information, see the description of the Export-Package header in the bundle manifest file.

2. Define bundle-type-specific metadata by adding the following headers to the bundle manifest file:

   Meta-Persistence
   The file path to the persistence.xml file.

Converting Java 2 security settings in an enterprise application to OSGi

When converting an enterprise archive (EAR) file to an enterprise bundle archive (EBA) file, you can have any Java 2 security settings converted automatically to OSGi, and then manually refine them to obtain the required OSGi security configuration.

In an enterprise application, the `was.policy` file defines Java 2 security permissions. In the `was.policy` file, you declare fine-grained security settings by using `grant codeBase` statements to grant permissions to application components.

In an OSGi application, permissions that define application-level security are specified in a `permissions.perm` file in the `META-INF` directory of the OSGi application. However, you can, in addition to the application-level `permissions.perm` file, create a `permissions.perm` file in the `OSGI-INF` directory of each bundle, to define finer-grained bundle-level access control.

When your Java 2 security settings in the `was.policy` are converted automatically, a `permissions.perm` file is created for you in the `META-INF` directory of the OSGi application. However, the automatic conversion process ignores `grant codeBase` statements, so that all the permissions in the `was.policy` file are copied to the `permissions.perm` file as application-level permissions. Therefore, you should, after conversion, review the permission settings in the `permissions.perm` file and move permissions to bundle-specific `permissions.perm` files as necessary.

To convert your Java 2 security settings to OSGi, complete the following steps:

1. Ensure you have completed the following conversion tasks, depending on the components of your EAR file:

- Change the file extension from .EAR to .EBA.
- Convert any web application archive (WAR) files to OSGi web application bundles.
- Convert any enterprise bean (EJB) Java archive (JAR) files to OSGi EJB bundles.
- Convert any utility JAR files to OSGi bundles.
- Convert any persistence archive files to OSGi bundles.

2. Import the EBA file as an asset. The Java 2 security settings are converted automatically during the import operation.
3. Export the EBA file to a location of your choice.
4. Open the `permissions.perm` file in the `META-INF` directory of the EBA file.
5. Review the permission settings in the `permissions.perm` file.
6. For those bundles that require bundle-specific permissions, create a `permissions.perm` file in the `OSGI-INF` directory of the bundle, and move the appropriate permissions to the newly-created file.
7. Update the previously imported EBA file with the modified EBA file.

Converting a Spring application to an OSGi application

To convert an application that is created using the Spring Framework to an OSGi application and move from the Spring Framework to standards-based technologies, you must modify the application manually. If a Spring application contains only web application archive (WAR) files, you can convert it automatically to run in OSGi Applications, but it still uses the Spring Framework.

About this task

In a Spring application, the Spring Framework manages features such as transactions, persistence, and dependency injection, and handles the communication between the servlets in the web container and the classes that handle the business logic of the application.

After you convert the application to an OSGi application that uses OSGi Applications support, the Blueprint Container manages the transactions, persistence and dependency injection.

If the application is an enterprise archive (EAR) file that contains only web application archive (WAR) files, you can convert it automatically. You might convert a Spring application in this way if a WAR file uses other library JAR files that must remain unchanged for the application to work. See “Converting a web application archive file to an OSGi web application bundle” on page 672.

Otherwise, you need to identify the Spring components and replace them with the equivalent code to make them plain old Java objects (POJOs), then modify the relevant XML files so that the Blueprint container manages those objects.

To convert a Spring application, you change it as follows:

- Create a well-defined interface to delegate to. The interface represents the classes that handle the business logic of the application.
- Create a service for each class, so that the servlet in the web container can use them.
- Change code that is specific to the Spring Framework in the classes to use the equivalent in Java EE, for example Java EE persistence classes.
- Create a handler servlet and interface to replace those from the Spring Framework.
- Modify XML and manifest files to use syntax that is correct for the Blueprint Container.

At the end of each step, the application is still usable, so you can decide whether to make all, or just some, of these changes.

The following procedure describes these steps in more detail.

Procedure

1. Create a well defined interface to delegate to, for example to replace a dispatcher servlet in the Spring application. The following example replaces the Spring HttpRequestHandler interface in an example application with the MyApplicationHandler interface.
   a. Create an interface to replace the Spring HttpRequestHandler interface.

```
package com.ibm.ws.eba.example.springconversion;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

interface MyApplicationHandler {

 void setMyApplicationHandle(MyApplicationUserInterface b);

 void handleRequest(HttpServletRequest request, HttpServletResponse response)
throws Exception;

}
```

   b. In each class that imports and implements the Spring HttpRequestHandler interface, implement the new interface. For example, change the following code:

```
implements HttpRequestHandler
```

      to :

```
implements MyApplicationHandler
```

   c. If the method creates an exception that is not in a base Java package, for example a ServletException exception, you can change it to use more general exception handling and avoid additional OSGi package imports. For example, change the following code:

```
handleRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
```

      to:

```
handleRequest(HttpServletRequest request, HttpServletResponse response)
throws Exception
```

   To replace the Spring interface with this new interface, you must also replace the servlet, as described in step 4 on page 679.
2. At this point, you could continue to use the Spring classes and use the Blueprint Container to manage those objects. To do this, change the `blueprint.xml` file, as described in step 7 on page 680.

   The configuration for dependency injection that the Blueprint Container uses is similar to the configuration for the Spring Framework. For example, if the Spring Framework calls the setMyApplicationHandle method to inject the *MyApplicationUserInterface* variable, this injection continues to work in the Blueprint Container, as long as the `blueprint.xml` file is configured correctly.
3. Change the managed bean that handles persistence to use standard Java EE persistence classes. In the following example, the Spring JpaTemplate interface handles persistence. It is equally valid if using JPA directly through an entity manager.
   a. Replace the JpaTemplate interface with Java EE persistence classes. For example, remove the following import statement:

```
import org.springframework.orm.jpa.JpaTemplate;
```

      Replace it with the following import statements:

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
```

   b. Use the relevant calls that are available in the API for those interfaces. For example, to replace the JpaTemplate code with EntityManager code, remove the following code:

```
@PersistenceUnit(unitName = "springExample")
  private JpaTemplate jpaTemplate;

  public void setJpaTemplate(JpaTemplate j)
  {
    jpaTemplate = j;
  }
```

Replace it with the following code:

```
@PersistenceContext(unitName = "myApplication")
  private EntityManager em;
  public void setEntityManager(EntityManager e) {
    em = e;
  }
```

c. If a persistence unit is inside a web application, separate the persistence unit from the WAR and create a persistence bundle. The persistence bundle must contain the entity classes and the `persistence.xml` file. The bundle manifest must have a Meta-Persistence header that points to the persistence unit, as shown in the following example:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyPersistenceBundle
Bundle-SymbolicName: com.ibm.ws.eba.example.persistence
Bundle-Version: 1.0.0
Meta-Persistence: WEB-INF/classes/META-INF/persistence.xml
Import-Package: javax.persistence
```

d. Replace each instance of a call from the bean that handles persistence with an equivalent call that uses the EntityManager interface, the Query interface, or both. For example, in the example Spring application, the following code snippet shows two methods that call the jpaTemplate.find method:

```
@SuppressWarnings("unchecked")
public boolean checkEMailAddressUniqueness(String emailAddress)
{
  boolean result = false;

  List<UserInfo> users = jpaTemplate.find(uniqueEmailAddressQuery, emailAddress);

  if (users.isEmpty()) {
    result = true;
  }

  return result;
}

public boolean checkUsernameUniqueness(String username)
{
  boolean result = false;

  UserInfo user = jpaTemplate.find(UserInfo.class, username);

  if (user == null) {
    result = true;
  }

  return result;
}
```

The following code snippet shows replacement code:

```
private static final String uniqueEmailAddressQuery = "select u from UserInfo u
where u.emailAddress = ?1";

@SuppressWarnings("unchecked")
public boolean checkEMailAddressUniqueness(String emailAddress)
{
  boolean result = false;
```

```
  Query q = em.createQuery(uniqueEmailAddressQuery);
  q.setParameter(1, emailAddress);

  List<UserInfo> users = q.getResultList();

  if (users.isEmpty()) {
    result = true;
  }

return result;
}

public boolean checkUsernameUniqueness(String username)
{
  boolean result = false;

  UserInfo user = em.find(UserInfo.class, username);

  if (user == null) {
    result = true;
  }

    return result;
  }
```

4. Create and register a handler servlet to replace the one from the Spring Framework.
   a. Create a servlet to forward requests to the appropriate handler in the Blueprint Container to process the business logic. This servlet is managed by the web container. The JNDI lookup is constructed as follows:

      `osgi:service/`*`class_name`*`/`*`ldap_filter`*

      The following code snippet creates the MyApplicationHandlerServlet servlet that passes an HTTP request to the MyApplicationHandler class in the Blueprint Container:

```
package com.ibm.ws.eba.example.springconversion;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyApplicationHandlerServlet extends HttpServlet {

 private static final long serialVersionUID = -3705932907251688199L;

 @Override
 public void service(HttpServletRequest request, HttpServletResponse response)
 {
  // Get servlet name
  String servletName = this.getServletConfig().getServletName();

  // Get business logic class from service registry that is
  // associated with the servlet

  try {
       InitialContext ic = new InitialContext();
       MyApplicationHandler myApplicationLogicClass = (MyApplicationHandler)
       ic.lookup("osgi:service/com.ibm.ws.eba.example.springconversion.
       MyApplicationHandler/(servlet.name="+servletName+")");
       myApplicationLogicClass.handleRequest(request, response);
   } catch (NamingException e) {
       e.printStackTrace();
   } catch (Exception e) {
           // TODO Auto-generated catch block
           e.printStackTrace();
  }
```

```
 }

}
```

b. Register the new servlet by changing the servlet classes to define the servlet you just created, rather than the Spring servlet. To do this, modify the `web.xml` file in the `WEB-INF` folder of the WAR file.

For example, change the following servlet definition in the `web.xml` file:

```
<servlet>
    <servlet-name>RegistrationServlet</servlet-name>
 <servlet-class> org.springframework.web.context.support.HttpRequestHandlerServlet
 </servlet-class>
  </servlet>
```

to:

```
<servlet>
    <servlet-name>RegistrationServlet</servlet-name>
 <servlet-class>com.ibm.ws.eba.example.springconversion.MyApplicationHandlerServlet
</servlet-class>
  </servlet>
```

c. Delete the following elements from the `web.xml` file:

```
<context-param>
<listener>
```

5. Separate the persistence unit from the EAR file and create a persistence bundle. The persistence bundle must contain the entity classes and the `persistence.xml` file. The bundle manifest must have a Meta-Persistence header that points to the persistence unit, as shown in the following example:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyPersistenceBundle
Bundle-SymbolicName: com.ibm.ws.eba.example.persistence
Bundle-Version: 1.0.0
Meta-Persistence: WEB-INF/classes/META-INF/persistence.xml
Import-Package: javax.persistence
```

6. If you removed the persistence unit from a WAR file, delete the following element from the `web.xml` file in the `WEB-INF` folder of the WAR file:

```
<persistence-unit-ref>
```

7. Change the XML so that the Blueprint Container manages the objects, rather than the Spring Framework.

a. Delete the `springapp-service.xml` file in the `WEB-INF` folder of the WAR file.

b. Create the `OSGI-INF/blueprint/` directory.

c. Create a file named `blueprint.xml` in the `OSGI-INF/blueprint/` directory. The following code shows an example `blueprint.xml` file:

```
<blueprint xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.0.0"
           xmlns:jpa="http://http://aries.apache.org/xmlns/jpa/v1.0.0">

  <bean id="myApplicationImpl"
    class="com.ibm.ws.eba.example.springconversion.impl.MyApplicationImpl">
    <jpa:context property="entityManager" unitname="myApplication" />
    <tx:transaction method="*" value="Required"/>
  </bean>

  <bean id="RegistrationServlet"
    class="com.ibm.ws.eba.example.springconversion.Registration">
    <tx:transaction method="*" value="Required"/>
    <property name="myApplicationHandle" ref="myApplicationService" />
  </bean>

  <service id="myApplicationService" ref="myApplicationImpl" interface=
```

```
        "com.ibm.ws.eba.example.springconversion.MyApplicationUserInterface" />

      <service id="registrationService" ref="RegistrationServlet" interface=
         "com.ibm.ws.eba.example.springconversion.MyApplicationHandler" >
        <service-properties>
          <entry key="servlet.name" value="RegistrationServlet"/>
        </service-properties>
      </service>

    </blueprint>
```

8. Update the `persistence.xml` file in the `war/WEB-INF/classes/META-INF/` directory. For example, you can change the JNDI lookups to use a service from a service registry. The following code is an example:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">

   <persistence-unit name="myApplication" transaction-type="JTA">
      <description>Persistence unit for the Spring conversion sample application
      </description>

      <provider>com.ibm.websphere.persistence.PersistenceProviderImpl
      </provider>

      <jta-data-source>osgi:service/javax.sql.DataSource/
        (osgi.jndi.service.name=jdbc/userdb)
      </jta-data-source>
      <non-jta-data-source>osgi:service/javax.sql.DataSource/
        (osgi.jndi.service.name=jdbc/userdbnonjta)
      </non-jta-data-source>
      <class>com.ibm.ws.eba.example.springconversion.Post</class>
      <class>com.ibm.ws.eba.example.springconversion.UserInfo</class>
      <exclude-unlisted-classes/>
   </persistence-unit>
</persistence>
```

9. Create the enterprise bundle archive (EBA) structure for the application.
   a. Create a `META-INF` directory in the root of the project and create an application manifest, `APPLICATION.MF`. The following code shows an example application manifest file:

```
Manifest-Version: 1.0
Application-ManifestVersion: 1.0
Application-Name: MyApplication
Application-SymbolicName: com.ibm.ws.eba.example.springconversion
Application-Version: 1.0
Application-Content: com.ibm.ws.eba.example.springconversion
```

   b. Delete the `context.xml` file from the `war/META-INF/` directory.
   c. In the `war/META-INF/` directory, create a bundle manifest file, `MANIFEST.MF`. The following code shows an example bundle manifest file:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyApplication
Bundle-SymbolicName: com.ibm.ws.eba.example.springconversion
Bundle-Version: 1.0.0
Bundle-Vendor: IBM
Bundle-ClassPath: WEB-INF/classes,
 WEB-INF/lib/aspectjrt.jar,WEB-INF/lib/aspectweaver.jar,
 WEB-INF/lib/commons-collections-3.2.jar,WEB-INF/lib/commons-lang-2.1.jar,
 WEB-INF/lib/JSON4J.jar,WEB-INF/lib/jstl.jar,WEB-INF/lib/jta.jar,
 WEB-INF/lib/openjpa-1.3.0-SNAPSHOT.jar,WEB-INF/lib/persistence.jar,
 WEB-INF/lib/serp-1.13.1.jar,WEB-INF/lib/standard.jar
Web-ContextPath: myApplication
```

```
Import-Package: javax.servlet;version="2.5",
 javax.servlet.http;version="2.5",
 javax.servlet.jsp;version="2.1",
 javax.servlet.jsp.tagext;version="2.1",
 org.osgi.framework;version="1.5.0",
 javax.persistence
```

Accessing Enterprise JavaBeans in OSGi applications

There are alternative mechanisms that you can use in a client bundle to access an enterprise bean in a service bundle. To maximize application availability during an update to a service bundle that contains one or more enterprise beans, use OSGi service references to access the enterprise beans.

From an OSGi client bundle, you can access an enterprise bean in a service bundle in any of the following ways:

- Define a <reference> element, in the Blueprint XML file of the client bundle, that refers to the enterprise bean.
- Use an @EJB annotation.
- Define an EJB reference that is mapped to the JNDI name of the bean.
- Use an "osgi:service/" JNDI lookup.

The alternative ways of accessing an enterprise bean, and the corresponding differences in application behaviour during an update to the service bundle, are discussed in more detail in the remaining sections of this topic.

Using a <reference> element to access an enterprise bean

You can access an enterprise bean by defining a reference element for the bean in the Blueprint XML file of the client bundle; for more information, see References and the Blueprint Container.

The enterprise bean must be in the list specified by the Export-EJB header in the bundle manifest file of the service bundle, or the value of the Export-EJB header must be set to a single space character so that all enterprise beans in the bundle are exported; for more information, see the description of the Export-EJB header in the bundle manifest file.

To access the enterprise bean from a blueprint-managed bean, you can inject the enterprise bean through a property on the blueprint-managed bean; for details, see References and the Blueprint Container.

To access the enterprise bean from a component that is not blueprint-managed, such as a servlet, use a "blueprint:comp/" JNDI lookup; for details, see JNDI lookup for blueprint components.

If you subsequently update the service bundle, the client bundle does not have to be restarted. During the update operation, an end user might experience a brief delay, because the service reference is damped, so the client bundle waits for the update operation to complete, rather than a runtime exception being thrown.

Using an @EJB annotation or an EJB reference to access an enterprise bean

You can create a reference to an enterprise bean by using an @EJB annotation in the source code of the client bundle, or by defining an EJB reference in an ejb-jar.xml or web.xml file and using a JNDI lookup in the source code. In either case, you map the reference to the enterprise bean JNDI name in an ibm-ejb-jar-bnd.xml or ibm-web-bnd.xml file in the bundle, or in the WebSphere Application Server administrative console when you deploy the bundle.

WebSphere Application Server automatically generates an enterprise bean JNDI name and maps the enterprise bean to it. In addition, you can explicitly map the enterprise bean to a JNDI name in the ibm-ejb-jar-bnd.xml file in the service bundle, or in the WebSphere Application Server administrative console when you deploy the bundle.

If you explicitly map the enterprise bean, and you subsequently update the service bundle, the client does not have to be restarted, but the service bundle is briefly unavailable, during which time a runtime exception is thrown if the client bundle attempts to invoke a method on the enterprise bean.

If you do not explicitly map the enterprise bean, and you subsequently update the service bundle, the JNDI name changes. The EJB reference to the JNDI name in the client bundle is updated automatically and the client bundle is restarted, during which time the client bundle is unavailable. In addition, the service bundle is briefly unavailable during the update operation, and if the client bundle becomes available before the service bundle and attempts to invoke a method on the enterprise bean, a runtime exception is thrown.

Using an "osgi:service/" lookup to access an enterprise bean

You can access an enterprise bean by using, in the source code of the client bundle, an "osgi:service/" lookup, in either of the following ways:

- Use a JNDI lookup with a URL of the following form:

  `osgi:service/`*`interface_name/optional_filter`*

  For example:

  ```
  try {
    InitialContext ic = new InitialContext();
    return (BloggingService)
      ic.lookup("osgi:service/com.ibm.samples.websphere.osgi.blog.api.BloggingService");
  } catch (NamingException e) {
      .
      .
      .
  ```

- Inject a reference to the enterprise bean by using an @Resource annotation of the following form:

  `@Resource(lookup="osgi:service/`*`interface_name/optional_filter`*`")`

  For example:

  ```
  @Resource(lookup="osgi:service/com.ibm.samples.websphere.osgi.blog.api.BloggingService")
  BloggingService bloggingService;
  ```

If you subsequently update the service bundle, the client bundle continues to run without interruption, but a ServiceUnavailableException is thrown if the code attempts to use the service object while the service bundle is being updated, so the source code must be written to handle this situation.

For details of the OSGi URL scheme, see section 126.6 of the OSGi Service Platform Release 4 Version 4.2 Enterprise Specification.

Summary

The following table summarizes the degree of availability during an update to an OSGi application that contains enterprise beans, depending on the enterprise bean access mechanism used:

Access mechanism	Summary of application availability
OSGi service reference	The client bundle remains available. The service reference is damped, so client threads block until the EJB service becomes available again.

Access mechanism	Summary of application availability
@EJB annotation or EJB reference, with explicit JNDI name mapping	The client bundle remains available. If the client bundle attempts to use the injected EJB, a runtime exception is thrown during the service bundle update.
"osgi:service/" JNDI lookup or injected reference	The client bundle remains available. If the client bundle attempts use the injected EJB, a runtime exception is thrown during the service bundle update. This behavior is similar to using an @EJB annotation or EJB reference with explicit JNDI name mapping, but more client code is required to achieve the same result.
@EJB annotation or EJB reference, without explicit JNDI name mapping	The client bundle is restarted. If the client bundle becomes available before the service bundle, attempts use the injected EJB will result in runtime exceptions being thrown until the service bundle update is complete.

Sample OSGi applications

OSGi Applications support includes sample applications that demonstrate how to write and package bundles into an enterprise bundle archive (EBA) file.

Two sample OSGi applications are provided: “Blog” and “Blabber”. Both applications demonstrate all of the following OSGi Applications features:

- Using Blueprint management.
- Using bean injection.
- Using services from the OSGi service registry.
- Publishing services to the OSGi service registry.
- Using the Java Persistence API (JPA).

The samples are provided in the `OSGi_blogSample.zip` and `OSGi_blabberSample.zip` compressed archive files, which can be downloaded from the Samples download page. The EBA file for each sample application is in the `installableApps` directory of the sample compressed file, and the source code is provided in other subdirectories of the sample compressed file.

For information about how to install and run each sample application, see the `sample_osgi_blog_readme` and `sample_osgi_blabber_readme` respectively. These files are available in the root directory of the associated sample compressed file. They are also available in the information center as topics “OSGi blog sample application” on page 685 and “OSGi blabber sample application” on page 690.

Blog sample application

The blog sample application is a traditional blogging application, used for publishing essay-length articles and allowing readers to comment on them. The application contains the following bundles:

- com.ibm.ws.eba.example.blog.persistence.

  This bundle contains JPA-related code, and the interfaces that enable the main application code to update and query blog entries.
- com.ibm.ws.eba.example.blog_1.0.0.

  This bundle contains the main application logic code and interacts between the web front end and the back end persistence code layer.
- com.ibm.ws.eba.example.blog.web.

  This bundle contains the static web content and backing Java code for the web front end of the application.
- com.ibm.ws.eba.example.blog.api.

This bundle contains the API for the sample.

- com.ibm.ws.eba.example.blog.persistence_1.1.0.

  This bundle contains an optional upgraded persistence bundle that also supplies a comment service.

To run the blog sample application, you can use all the bundles that are listed, or all the bundles except the upgraded persistence bundle (the last in the list).

For more information, see “OSGi blog sample application.”

Blabber sample application

The blabber sample application is a microblogging application, used to enable many different people to share brief comments. The application contains the following bundles:

- com.ibm.ws.eba.example.blabber.persistence.

  This bundle contains code that relates to the Java Persistence API (JPA) layer and the interface for the main application code.

- com.ibm.ws.eba.example.blabber.

  This bundle contains the main application code and the code for the web front end of the application.

For more information, see “OSGi blabber sample application” on page 690.

OSGi blog sample application

The blog sample application is a traditional blogging application, used for publishing essay-length articles and allowing readers to comment on them. This sample application shows how to write and package bundles into an enterprise bundle archive (EBA) file. The sample includes example code for Blueprint management, bean injection, using and publishing services from and to the OSGi service registry, and the use of Java persistence.

Before you begin

The blog sample application consists of five bundles, but can be run with four bundles because the fifth bundle is an upgrade to the persistence bundle.

The sample application requires that the supplied `com.ibm.samples.websphere.osgi.logging.api.jar` and `com.ibm.samples.websphere.osgi.logging.impl.jar` bundles are installed into the internal bundle repository. The following procedure describes how to do this. These two JAR files, and the blog sample EBA file, are provided in the `installableApps` directory of the `OSGi_blogSample.zip` compressed archive file.

About this task

The bundles are divided into the following functional areas:

- `com.ibm.samples.websphere.osgi.blog.persistence`, which contains code relating to the Java Persistence API (JPA) layer, and also contains interfaces that are used by the main application code to update and query blog entries.
- `com.ibm.samples.websphere.osgi.blog_1.0.0`, which contains the main application logic code, and interacts between the web front end and the back end persistence code layer.
- `com.ibm.samples.websphere.osgi.blog.web`, which contains the static web content and backing Java code for the web front end for the application.
- `com.ibm.samples.websphere.osgi.blog.api`, which contains the API for the whole sample.
- `com.ibm.samples.websphere.osgi.blog.persistence_1.1.0`, which contains an upgraded persistence bundle that also supplies a comment service.

You can use scripts to completely install the sample application, or you can use scripts to complete the initial configuration of the application then use the administrative console to install the application into the application server. You can also use, modify and remove the sample, and upgrade the persistence service that is provided by the sample.

Procedure

Configure and install the blog sample.
You can either configure and install the sample using scripts (the first optional step in the instructions that follow), or you can configure the sample using scripts then install the sample using the administrative console (the second optional step in the instructions that follow).
The scripts that you use to do this are provided in the `scripts` directory of the `OSGi_blogSample.zip` compressed archive file. The `createBlogDb.sql` script contains the necessary configuration commands to create the required Derby database and associated tables. The `blogSampleInstall.py` script contains the required jython to create data sources, and to install the blog sample with default configuration. You should fully qualify the path to the script if you do not run it from the directory that contains the script.

Note: If you have data sources already defined in your environment, and these data sources have the same name as the data sources that are specified in the `blogSampleInstall.py` script, then the sample might not install and run successfully.

In the following steps you must substitute your own values for the variables *app_server_root*, *profileName*, *serverName*, *nodeName*, *blogSample.eba_Location*, *com.ibm.samples.websphere.osgi.logging.api.jar_Location*, *com.ibm.samples.websphere.osgi.logging.impl.jar_Location*, *uncompressed_sample_directory*, and *path_to_ant*.

- Optional: Configure and install the sample using scripts.
  1. Create and configure the “BLOGDB” Derby database and associated tables by running the following command.

     On UNIX platforms:

     ```
     app_server_root/derby/bin/embedded/ij.sh scripts/createBlogDb.sql
     ```

     On Windows platforms:

     ```
     app_server_root\derby\bin\embedded\ij.bat scripts\createBlogDb.sql
     ```

  2. Create the data sources, create the business-level application and import the .eba file as an asset by running the following command.

     Note: The target application server must be running before you use this script.

     On UNIX platforms:

     ```
     app_server_root/profiles/profileName/bin/wsadmin.sh
          -f scripts/blogSampleInstall.py fullInstall serverName nodeName
          blogSample.eba_Location
          com.ibm.samples.websphere.osgi.logging.api.jar_Location
          com.ibm.samples.websphere.osgi.logging.impl.jar_Location
     ```

     On Windows platforms:

     ```
     app_server_root\profiles\profileName\bin\wsadmin.bat
          -f scripts\blogSampleInstall.py fullInstall serverName nodeName
          blogSample.eba_Location
          com.ibm.samples.websphere.osgi.logging.api.jar_Location
          com.ibm.samples.websphere.osgi.logging.impl.jar_Location
     ```

- Optional: Configure the sample using scripts, then install the sample using the administrative console.
  1. Create and configure the “"BLOGDB" ” Derby database and associated tables by running the following command.

     On UNIX platforms:

     ```
     app_server_root/derby/bin/embedded/ij.sh scripts/createBlogDb.sql
     ```

     On Windows platforms:

app_server_root\derby\bin\embedded\ij.bat scripts\createBlogDb.sql

2. Create the data sources by running the following command.

   On UNIX platforms:

   app_server_root/profiles/*profileName*/bin/wsadmin.sh
   -f scripts/blogSampleInstall.py setupOnly *serverName nodeName*

   On Windows platforms:

   app_server_root\profiles*profileName*\bin\wsadmin.bat
   -f scripts\blogSampleInstall.py setupOnly *serverName nodeName*

3. Use the administrative console to add the com.ibm.samples.websphere.osgi.logging.api.jar file to the internal bundle repository.
   a. Navigate to **Environment** > **OSGi bundle repositories** > **Internal bundle repository**
   b. Click **New**.
   c. Select the file system that hosts the com.ibm.samples.websphere.osgi.logging.api.jar file, then click **Browse**.
   d. Browse to the *uncompressed_sample_dir*/installableApps directory.
   e. Select the com.ibm.samples.websphere.osgi.logging.api.jar file, then click **OK**.
   f. Click **OK**.
   g. Click **Save**.
4. Repeat the previous step to add the com.ibm.samples.websphere.osgi.logging.impl.jar file to the internal bundle repository.
5. Use the administrative console to import the application (EBA file) as an asset, and to configure the business-level application.
   a. Import the asset.
      1) Navigate to **Applications** > **New application** > **New Asset**.
      2) On the Upload asset panel, browse to the *uncompressed_sample_dir*/installableApps directory.
      3) Select the com.ibm.samples.websphere.osgi.blog.eba file, then click **Next**.
      4) On the Select options for importing an asset panel, click **Next**.
      5) On the Summary panel, click **Finish**.
      6) Click **Save**.
   b. Create the business-level application and add the asset.
      1) Navigate to **Applications** > **New application** > **New Business Level Application**.
      2) On the New Application panel, enter a name for the business-level application. For example, “blog”.
      3) Click **Apply**.
      4) In the Deployed assets pane, click **Add** > **Add Asset**.
      5) Select the com.ibm.samples.websphere.osgi.blog.eba asset, then click **Continue**.
      6) On the Set options panel, click **Next**.
      7) Select a target for the composition unit, then click **Next**.
      8) Modify the context root if required, then click **Next**.
      9) Modify the virtual host if required, then click **Next**.
      10) On the Summary panel, click **Finish**.
      11) Click **Save**.
6. Start the application.
   a. Select the newly-created business-level application.
   b. Click **Start**.

Use the blog sample.

- Use your browser to navigate to `http://server:port/context_root/` (by default `http://localhost:9080/blog/`). The View blog screen is displayed.
- Register yourself as an Author. You need to do this before you can create blog entries.
  1. Click **Create Author**.
  2. Complete the form.
  3. Click **Submit**.
- Create a blog entry.
  1. Navigate back to `http://localhost:9080/blog/` (or click **Blog Home**).
  2. Click **Create New Post**.
  3. Complete the form. Use the email address that you specified when you created the Author.
  4. Click **Submit**.

  The View blog screen is redisplayed, and should contain your post.

Upgrade the persistence service.

To add the new persistence bundle to the blog application, use the administrative console to load the `com.ibm.samples.websphere.osgi.blog.persistence_1.1.0.jar` file into the internal bundle repository, then modify the asset to expect a new bundle version. Complete the following steps:

- Load the persistence bundle into the internal bundle repository.
  1. Navigate to **Environment** > **OSGi bundle repositories** > **Internal bundle repository**.
  2. Click **New**.
  3. Browse to the *uncompressed_sample_dir*`/installableApps` directory.
  4. Select the `com.ibm.samples.websphere.osgi.blog.persistence_1.1.0.jar` file, then click **OK**.
  5. Save your changes to the master configuration.
- Add the persistence bundle to the blog application.
  1. Navigate to **Applications** > **Application types** > **Assets**.
  2. Click **com.ibm.samples.websphere.osgi.blog.eba** to view the asset information.
  3. Scroll to the end of the asset information, then click **Update bundle versions in this application**.
  4. Find the `com.ibm.samples.websphere.osgi.blog.persistence` bundle in the list.
  5. Click the down arrow beside the text “no preference”. You are offered a choice between version 1.0.0 or 1.1.0 of the persistence bundle.
  6. Select version 1.1.0, then work through the preview, create and save screens.
- Update the application to use the new bundle version.
  1. Navigate to **Applications** > **Application types** > **Business-level applications**.
  2. Click the blog sample application to display its details.
  3. Click **com.ibm.samples.websphere.osgi.blog.eba** to view the asset information.
  4. Click **Update to latest deployment**, then click **OK**.
  5. Save your changes to the master configuration.
- Stop and restart the blog application.
  1. Navigate to **Applications** > **Application types** > **Business-level applications**.
  2. Select the check box alongside the blog sample application, then click **Stop**.
  3. Wait until the status column shows that the application has stopped, then click **Start**.
  4. Use your browser to navigate to `http://server:port/context_root/` (by default `http://localhost:9080/blog/`) then refresh the screen. The View blog screen is displayed. There is a new option to add comments to blog posts.

Modify the blog sample

All the source code for this application is provided in sub-directories below the *uncompressed_sample_dir* directory. Each part of the project has its own ant `build.xml` script. To build the whole application into a newly-available EBA file, you use the ant `build.xml` file located in the *uncompressed_sample_dir* directory. WebSphere Application Server ships a version of ant in its `bin` directory called `ws_ant`. To build the sample, you also need a JAR file on the ant classpath. To simplify matters, you can edit the **was.root** property in the supplied `build.properties` file to point to this JAR file.

- Put the JAR file `j2ee.jar` on the ant classpath:

  This JAR file is available in the `dev/JavaEE` directory of Websphere Application Server. Edit the **was.root** property in the supplied *uncompressed_sample_dir*`/scripts/build.properties` file, and point it to your *app_server_root* directory. If you have copied the JAR file to another location, modify the **was.root** property to point to this location.
- Run the following ant command from the *uncompressed_sample_dir* directory:

  path_to_ant `-propertyfile scripts/build.properties -buildfile build.xml`

  The newly-built binary file (EBA file) is in the *uncompressed_sample_dir*`/output` directory.

  Note: This is the only location to which this EBA file is written. The script does not overwrite the original binary file located in the *uncompressed_sample_dir*`/installableApps` directory.

Remove the blog sample.

To remove the blog sample, you complete the following 3 steps:

1. Remove the application configuration and the data sources, either by using a script (the first optional step in the instructions that follow), or by using the administrative console (the second optional step in the instructions that follow).
2. Remove the shared logging bundles from the internal bundle repository.
3. Remove the database.

Note: Only remove the bundles if you have no other applications installed that use them. Both the blog and blabber sample applications use the shared logging bundles, so if both applications are installed these bundles should not be removed.

The uninstall script is provided in the `scripts` directory of the `OSGi_blogSample.zip` compressed archive file. The `blogSampleUninstall.py` script contains the required jython to remove data sources and to remove the installation of the blog sample with default configuration. You should fully qualify the path to the script if you do not run it from the directory that contains the script.

In the following steps you must substitute your own values for the variables *app_server_root*, and *profileName*.

- Optional: Remove the application configuration and the data sources by running the following command.

  Note: The target application server must be running before you use this script.

  On UNIX platforms:

  ```
  app_server_root/profiles/profileName/bin/wsadmin.sh
            -f scripts/blogSampleUninstall.py
  ```

  On Windows platforms:

  ```
  app_server_root\profiles\profileName\bin\wsadmin.bat
            -f scripts\blogSampleUninstall.py
  ```

- Optional: Remove the application configuration and the data sources by using the administrative console.
  1. Remove the application configuration.
     a. Navigate to **Applications** > **Application types** > **Business-level applications**.
     b. Select the business-level application representing the blog sample application, then click **Stop**.

c. Click the business-level application representing the blog sample application to view the configuration details.
d. Select all the deployed assets, then click **Delete**.
e. Click **OK** to confirm removal.
f. Click **Save**.
g. Select the business-level application representing the blog sample application, then click **Delete**.
h. Navigate to **Applications** > **Application types** > **Assets**.
i. Select the `com.ibm.samples.websphere.osgi.blog.eba` asset, then click **Delete**.
j. Click **OK** to confirm removal.
k. Click **Save**.

2. Remove the data sources.
   a. Navigate to **Resources** > **JDBC** > **Data sources**.
   b. Select the two data sources configured for the blog application.
   c. Click **Delete**.
   d. Click **Save**.

- Remove the shared logging bundles from the internal bundle repository by using the administrative console.
  1. Navigate to **Environment** > **OSGi bundle repositories** > **Internal bundle repository**.
  2. Select the following bundles:
     - `com.ibm.samples.websphere.osgi.logging.api.jar`
     - `com.ibm.samples.websphere.osgi.logging.impl.jar`
     - `com.ibm.samples.websphere.osgi.blog.persistence_1.1.0`
  3. Click **Delete**.
  4. Click **Save**.
- Remove the database.

  On UNIX platforms, open a command prompt, then enter the following command:

  ```
  cd app_server_root/derby
  rm -fr BLOGDB
  ```

  On Windows platforms, delete the *app_server_root*`\derby\BLOGDB` directory.

OSGi blabber sample application

The blabber sample application is a microblogging application, used to enable many different people to share brief comments. This sample application shows how to write and package bundles into an enterprise bundle archive (EBA) file. The sample includes example code for Blueprint management, bean injection, using and publishing services from and to the OSGi service registry, and the use of Java persistence.

Before you begin

The sample application requires that the supplied `com.ibm.samples.websphere.osgi.logging.api.jar` and `com.ibm.samples.websphere.osgi.logging.impl.jar` bundles are installed into the internal bundle repository. The following procedure describes how to do this. These two JAR files, and the blabber sample EBA file, are provided in the `installableApps` directory of the `OSGi_blabberSample.zip` compressed archive file.

About this task

The blabber sample application consists of two bundles, divided into the following functional areas:

- `com.ibm.samples.websphere.osgi.blabber.persistence`, which contains code relating to the Java Persistence API (JPA) layer, and also contains the interface that is used by the main application code.

- `com.ibm.samples.websphere.osgi.blabber`, which contains the main application logic code, and interacts between the web front end and the back end persistence code layer.

You can use scripts to completely install the sample application, or you can use scripts to complete the initial configuration of the application then use the administrative console to install the application into the application server. You can also use, modify and remove the sample.

Procedure

Configure and install the blabber sample.
You can either configure and install the sample using scripts (the first optional step in the instructions that follow), or you can configure the sample using scripts then install the sample using the administrative console (the second optional step in the instructions that follow).
The scripts that you use to do this are provided in the `scripts` directory of the `OSGi_blabberSample.zip` compressed archive file. The `createblabberDb.sql` script contains the necessary configuration commands to create the required Derby database and associated tables. The `blabberSampleInstall.py` script contains the required jython to create data sources, and to install the blabber sample with default configuration. You should fully qualify the path to the script if you do not run it from the directory that contains the script.

Note: If you have data sources already defined in your environment, and these data sources have the same name as the data sources that are specified in the `blabberSampleInstall.py` script, then the sample might not install and run successfully.

In the following steps you must substitute your own values for the variables *app_server_root*, *profileName*, *serverName*, *nodeName*, *blabberSample.eba_Location*, *com.ibm.samples.websphere.osgi.logging.api.jar_Location*, *com.ibm.samples.websphere.osgi.logging.impl.jar_Location*, *uncompressed_sample_directory*, and *path_to_ant*.

- Optional: Configure and install the sample using scripts.
  1. Create and configure the “BLABBERDB” Derby database and associated tables by running the following command.

     On UNIX platforms:

     ```
     app_server_root/derby/bin/embedded/ij.sh scripts/createBlabberDb.sql
     ```

     On Windows platforms:

     ```
     app_server_root\derby\bin\embedded\ij.bat scripts\createBlabberDb.sql
     ```

  2. Create the data sources, create the business-level application and import the .eba file as an asset by running the following command.

     Note: The target application server must be running before you use this script.

     On UNIX platforms:

     ```
     app_server_root/profiles/profileName/bin/wsadmin.sh
          -f scripts/blabberSampleInstall.py fullInstall serverName nodeName
          blabberSample.eba_Location
          com.ibm.samples.websphere.osgi.logging.api.jar_Location
          com.ibm.samples.websphere.osgi.logging.impl.jar_Location
     ```

     On Windows platforms:

     ```
     app_server_root\profiles\profileName\bin\wsadmin.bat
          -f scripts\blabberSampleInstall.py fullInstall serverName nodeName
          blabberSample.eba_Location
          com.ibm.samples.websphere.osgi.logging.api.jar_Location
          com.ibm.samples.websphere.osgi.logging.impl.jar_Location
     ```

- Optional: Configure the sample using scripts, then install the sample using the administrative console.
  1. Create and configure the “"BLABBERDB" ” Derby database and associated tables by running the following command.

On UNIX platforms:

```
app_server_root/derby/bin/embedded/ij.sh scripts/createBlabberDb.sql
```

On Windows platforms:

```
app_server_root\derby\bin\embedded\ij.bat scripts\createBlabberDb.sql
```

2. Create the data sources by running the following command.

   On UNIX platforms:

   ```
   app_server_root/profiles/profileName/bin/wsadmin.sh
        -f scripts/blabberSampleInstall.py setupOnly serverName nodeName
   ```

   On Windows platforms:

   ```
   app_server_root\profiles\profileName\bin\wsadmin.bat
        -f scripts\blabberSampleInstall.py setupOnly serverName nodeName
   ```

3. Use the administrative console to add the `com.ibm.samples.websphere.osgi.logging.api.jar` file to the internal bundle repository.
   a. Navigate to **Environment** > **OSGi bundle repositories** > **Internal bundle repository**
   b. Click **New**.
   c. Select the file system that hosts the `com.ibm.samples.websphere.osgi.logging.api.jar` file, then click **Browse**.
   d. Browse to the *uncompressed_sample_dir*`/installableApps` directory.
   e. Select the `com.ibm.samples.websphere.osgi.logging.api.jar` file, then click **OK**.
   f. Click **OK**.
   g. Click **Save**.
4. Repeat the previous step to add the `com.ibm.samples.websphere.osgi.logging.impl.jar` file to the internal bundle repository.
5. Use the administrative console to import the application (EBA file) as an asset, and to configure the business-level application.
   a. Import the asset.
      1) Navigate to **Applications** > **New application** > **New Asset**.
      2) On the Upload asset panel, browse to the *uncompressed_sample_dir*`/installableApps` directory.
      3) Select the `com.ibm.samples.websphere.osgi.blabber.app.eba` file, then click **Next**.
      4) On the Select options for importing an asset panel, click **Next**.
      5) On the Summary panel, click **Finish**.
      6) Click **Save**.
   b. Create the business-level application and add the asset.
      1) Navigate to **Applications** > **New application** > **New Business Level Application**.
      2) On the New Application panel, enter a name for the business-level application. For example, “blabber”.
      3) Click **Apply**.
      4) In the Deployed assets pane, click **Add** > **Add Asset**.
      5) Select the `com.ibm.samples.websphere.osgi.blabber.app.eba` asset, then click **Continue**.
      6) On the Set options panel, click **Next**.
      7) Select a target for the composition unit, then click **Next**.
      8) Modify the context root if required, then click **Next**.
      9) Modify the virtual host if required, then click **Next**.
      10) On the Summary panel, click **Finish**.
      11) Click **Save**.
6. Start the application.

a. Select the newly-created business-level application.
b. Click **Start**.

Use the blabber sample.

- Use your browser to navigate to `http://server:port/context_root/` (by default `http://localhost:9080/blabber/`). The main blabber screen is displayed.
- Register yourself as a user. You need to do this before you can create blabber entries.
  1. Click **Click here to sign-up!**.
  2. Complete the form.
  3. Click **Create my account!**.

  The status update home page is displayed.
- Explore the blabber application.

  From the status update home page, you can select any of the following options:
  – Set a status message.
  – Search for and track other users.
  – View your profile.
  – Sign out.

  Within each of the previous options, there is an option to return to the status update home page.

  Note:
  – When you enter some status then click **Update**, your status message is displayed on the current page.
  – You can only view or use the tracking option if the application has more than one user.

Modify the blabber sample.

All the source code for this application is provided in sub-directories the *uncompressed_sample_dir* directory. Each part of the project has its own ant `build.xml` script. To build the whole application into a newly-available EBA file, you use the ant `build.xml` file located in the *uncompressed_sample_dir* directory. WebSphere Application Server ships a version of ant in its `bin` directory called `ws_ant`. To build the sample, you also need several JAR files on the ant classpath. To simplify matters, you can edit the **was.root** property in the supplied `build.properties` file to point to these JAR files.

- Put the following jar files on the ant classpath:
  – `javax.j2ee.persistence.jar`
  – `javax.j2ee.jsp.jar`
  – `javax.j2ee.servlet.jar`
  – `com.ibm.ws.prereq.jaxrs.jar`

  These JAR files are available in the `plugins` directory of Websphere Application Server. Edit the **was.root** property in the supplied *uncompressed_sample_dir*`/scripts/build.properties` file, and point it to your *app_server_root* directory. If you have copied the jars to another location, modify the **was.root** property to point to this location.
- Run the following ant command from the *uncompressed_sample_dir* directory:

  path_to_ant `-propertyfile scripts/build.properties -buildfile build.xml`

  The newly-built binary file (EBA file) is in the *uncompressed_sample_dir*`/output` directory.

  Note: This is the only location to which this EBA file is written. The script does not overwrite the original binary file located in the *uncompressed_sample_dir*`/installableApps` directory.

Remove the blabber sample.

To remove the blabber sample, you complete the following 3 steps:

1. Remove the application configuration and the data sources, either by using a script (the first optional step in the instructions that follow), or by using the administrative console (the second optional step in the instructions that follow).
2. Remove the shared logging bundles from the internal bundle repository.
3. Remove the database.

Note: Only remove the bundles if you have no other applications installed that use them. Both the blabber and blog sample applications use the shared logging bundles, so if both applications are installed these bundles should not be removed.

The uninstall script is provided in the `scripts` directory of the `OSGi_blabberSample.zip` compressed archive file. The `blabberSampleUninstall.py` script contains the required jython to remove data sources and to remove the installation of the blabber sample with default configuration. You should fully qualify the path to the script if you do not run it from the directory that contains the script.

In the following steps you must substitute your own values for the variables *app_server_root*, and *profileName*.

- Optional: Remove the application configuration and the data sources by running the following command.

  Note: The target application server must be running before you use this script.

  On UNIX platforms:

  ```
  app_server_root/profiles/profileName/bin/wsadmin.sh
           -f scripts/blabberSampleUninstall.py
  ```

  On Windows platforms:

  ```
  app_server_root\profiles\profileName\bin\wsadmin.bat
           -f scripts\blabberSampleUninstall.py
  ```

- Optional: Remove the application configuration and the data sources by using the administrative console.
  1. Remove the application configuration.
     a. Navigate to **Applications** > **Application types** > **Business-level applications**.
     b. Select the business-level application representing the blabber sample application, then click **Stop**.
     c. Click the business-level application representing the blabber sample application to view the configuration details.
     d. Select all the deployed assets, then click **Delete**.
     e. Click **OK** to confirm removal.
     f. Click **Save**.
     g. Select the business-level application representing the blabber sample application, then click **Delete**.
     h. Navigate to **Applications** > **Application types** > **Assets**.
     i. Select the `com.ibm.samples.websphere.osgi.blabber.app.eba` asset, then click **Delete**.
     j. Click **OK** to confirm removal.
     k. Click **Save**.
  2. Remove the data sources.
     a. Navigate to **Resources** > **JDBC** > **Data sources**.
     b. Select the two data sources configured for the blabber application.
     c. Click **Delete**.
     d. Click **Save**.

- Remove the shared logging bundles from the internal bundle repository by using the administrative console.
  1. Navigate to **Environment** > **OSGi bundle repositories** > **Internal bundle repository**.
  2. Select the following bundles:
     - `com.ibm.samples.websphere.osgi.logging.api.jar`
     - `com.ibm.samples.websphere.osgi.logging.impl.jar`
  3. Click **Delete**.
  4. Click **Save**.
- Remove the database.

  On UNIX platforms, open a command prompt, then enter the following command:

  ```
  cd app_server_root/derby
  rm -fr BLABBERDB
  ```

  On Windows platforms, delete the *app_server_root*\derby\BLABBERDB directory.

Chapter 18. Developing Portlet applications

This page provides a starting point for finding information about portlet applications, which are special reusable Java servlets that appear as defined regions on portal pages. Portlets provide access to many different applications, services, and web content.

Portlet aggregation and preferences

Supported optional features of the JSR-286 Portlet Specification

The simple portal framework, which builds on top of the portlet container, is JSR-286-compliant. However, the WebSphere Application Server implementation supports a subset of the optional features in the JSR-286 Portlet Specification.

The following table lists the optional features that are available in the JSR-286 Portlet Specification and indicates to what extent these optional features are available in WebSphere Application Server.

Table 83. Optional features in the JSR-286 Portlet Specification. The Availability column describes whether the product supports a specification feature.

Feature	Availability
Container runtime options	The following container runtime options are supported: • javax.portlet.escapeXml • javax.portlet.servletDefaultSessionScope • javax.portlet.actionScopedRequestAttributes
Setting the HTML head section elements using the MimeResponse.MARKUP_HEAD_ELEMENT property	This property is not supported as its setting does not have an effect.
Custom portlet modes and custom window states	The feature is supported, but without special treatment.
Portlet-managed modes	This feature is fully supported.
Dynamically setting the portlet title using the RenderResponse.setTitle(String) method	This feature is partially supported. You need to make use of this feature in a portlet document filter or an aggregation JavaServer Pages (JSP) file.
Dynamically setting the next possible portlet modes using the RenderResponse.setNextPossiblePortletModes(Collection<PortletMode>) method	This feature is not supported.
Expiration and validation-based caching	This feature is fully supported. You need to activate the portlet fragment caching and a cachespec.xml defined as a prerequisite.
Aliases in public render parameters	The aliases in public render parameters are supported in the PortletServingServlet servlet and the aggregation tag library.
Aliases and wild cards in eventing	These aliases and wild cards are not supported as setting these functions does not have an effect.

Aggregation tag library attributes

The aggregation tag library is used to aggregate multiple portlets on one page. This topic describes the attributes within the aggregation tag library.

Supported arguments include:

`init`

This tag initializes the portlet framework and has to be used in the beginning of the JSP. All other tags described in this section are only valid in the body of this tag, therefore the init tag usually encloses the whole body of a JSP. In case the current URL contains an action flag the action method of the corresponding portlet is called. The state and insert tags are sub-tags of the init tag.

The init tag has the following attributes:

- portletURLPrefix = ”<any string>”

This URL defines the prefix used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet's render method, which is called by using the insert tag. This is a required attribute.

- portletURLSuffix = ”<any string>”

  This URL defines the suffix used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet's render method, which is called by using the insert tag. This is attribute optional.

- portletURLQueryParams = ”<any string>”

  This URL defines the query parameters used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet's render method, which is called by using the insert tag. This is attribute optional.

scope, portlet

The scope tag and portlet tag are used to provide information that is necessary when a portlet application is installed under a multiple part context root, for example, /context1/context2. These tags also define which portlet windows should participate in portlet coordination via public render parameters, and add a render parameter to the newly created URL.

The urlParam tag has the following attributes:

- context = "/<context1>/<context2>"

  Specifies the context root of the portlet application in which the portlet is deployed. This attribute is required.

- portletname = "<portlet-name>"

  Specifies the portlet-name. This attribute is required.

- windowId = ”<any string>”

  Defines the window ID for the concrete portlet instance. This attribute is required.

The following is an example of how to use the scope and portlet tags:

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>

<portlet:scope>
    <portlet:portlet context="/myportletcontext1/myportletcontext2" portletname="MyPortlet" windowId="sample"/>
</portlet:scope>

<portlet:init portletURLPrefix="/myportalcontext/ ">
....
</portlet:init>
```

state

The state tag creates a URL pointing to the given portlet using the given state. You can place this URL either into a variable specified by the var attribute or you can write it directly to the output stream. This tag is useful to create URLs for HTML buttons, images, and other items such that when the URL is invoked, the state changes defined in the URL are applied to the given portlet.

The state tag has the following attributes:

- url = ”<context>/<portlet-name>”

  Identifies the portlet for this tag by using the context and portlet-name to address the portlet. This attribute is required.

- windowId = ”<any string>”

  Defines the window ID for the portlet URL created by this tag. This is attribute optional.

- var = ”<any string>”

  If defined the URL is written into a variable with the given scope and name, not to the output stream. This is attribute optional.

- scope = ”page|request|session|application”

  This attribute is only valid if the var attribute is specified. If defined, the URL is not written to the output stream but a variable is created in the given scope with the given name. The default is page. This is attribute optional.

- portletMode = ”view|help|edit|<custom>”

  This attribute sets the portlet mode.
- portletWindowState = ”maximized|minimized|normal|<custom>”

  This attribute sets the window state.
- action = ”true/false”

  This attribute defines whether this is an action URL. This is attribute optional. The default is false.

urlParam

Adds a render parameter to the newly created URL.

The urlParam tag has the following attributes:

- name = ”<any string>”

  Indicates the name of the parameter. This is attribute required.
- value = ”<any string>”

  Indicates the value of the parameter. This is attribute required.

insert

This tag calls the render method of the portlet and retrieves the content as well as the title. You can optionally place the content and title of the specified portlet into variables using the contentVar and titleVar attributes.

The insert tag has the following attributes:

- url = ”<context>/<portlet-name>” (mandatory) Identifies the portlet for this tag by using the context and portlet-name to address the portlet

  This is attribute required.
- windowId = ”<any string>”

  Defines the window ID of the portlet. This is attribute optional.
- contentVar = ”<any string>”

  If defined, the portlet's content is not written to the output stream but written into a variable with the given scope and name. This is attribute optional.
- contentScope = ”page|request|session|application”

  This attribute is only valid if the contentVar tag is used. If defined, the portlet's content is written into a variable with the given scope and name, not to the output stream. The default is page. This is attribute optional.
- titleVar = ”<any string>”

  If defined the portlet's title is written into a variable with the given scope and name. If it is not defined, the title is ignored and not written to the output stream. This is attribute optional.
- titleScope = ”page|request|session|application”

  This attribute is only valid if titleVar tag is used. If defined, the portlet's title is written into a variable with the given scope and name, not to the output stream. The default is page. This is attribute optional.

Example: Using the portlet aggregation tag library

You can use the aggregation tag library to aggregate multiple portlets to have multiple and different content on one page. The library can be used by every JavaServer Pages (JSP) file that has been included by a servlet.

To use the portlet aggregation tag library, you must declare the tag-lib at the top of the JSP file using, `<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>`, as in the following example. The following JSP file example shows how to aggregate portlets on one page.

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>
<%@ page isELIgnored="false" import="java.util.Enumeration"%>

<portlet:init portletURLPrefix="/dummy/portletTagTest/" portletURLSuffix="/more" portletURLQueryParams="p1=v1&p2=v2">
```

```
   <portlet:insert url="worldclock/StdWorldClock" contentVar="worldclockcontent" titleVar="worldclocktitle"/>
    <portlet:state url="worldclock/StdWorldClock" portletMode="view" var="worldclockview"
       portletWindowState="maximized">
        <portlet:urlParam name="namea" value="valuea"/>
        <portlet:urlParam name="nameb" value="valueb"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" portletMode="edit" var="worldclockedit" portletWindowState="normal">
        <portlet:urlParam name="name1" value="value1"/>
        <portlet:urlParam name="name2" value="value2"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" portletMode="view" var="worldclockmin"
       portletWindowState="minimized">
        <portlet:urlParam name="namemin" value="valuemin"/>
        <portlet:urlParam name="namemin" value="valuemin"/>
    </portlet:state>

    <portlet:insert url="worldclock/StdWorldClock" windowId="min" contentVar="simplecontent" titleVar="simpletitle"/>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="view" var="simpleview"
  portletWindowState="maximized">
        <portlet:urlParam name="name3" value="value3"/>
        <portlet:urlParam name="name4" value="value4"/>
        <portlet:urlParam name="name5" value="value5"/>
        <portlet:urlParam name="name5" value="value5a"/>
        <portlet:urlParam name="name5" value="value5b"/>
        <portlet:urlParam name="name5" value="value5c"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="edit" var="simpleedit"
  action="true" portletWindowState="normal">
        <portlet:urlParam name="name6" value="value6"/>
        <portlet:urlParam name="name6" value="value6z"/>
            </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="view" var="simplemin"
  portletWindowState="minimized">
        <portlet:urlParam name="name1" value="value1"/>
        <portlet:urlParam name="name2" value="value2"/>
    </portlet:state>

    <portlet:insert url="test/TestPortlet1" contentVar="testcontent" titleVar="testtitle"/>
    <portlet:state url="test/TestPortlet1" portletMode="view" var="testview" portletWindowState="maximized"/>
    <portlet:state url="test/TestPortlet1" portletMode="edit" var="testedit" portletWindowState="maximized"/>


<!--  This table is the outtermost table for creating two-column portal layout  -->
<TABLE border="0" CELLPADDING="3" CELLSPACING="8" WIDTH="100%">
<TR>
<TD VALIGN="top">

<!--  This table is the top portlet in the first column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet top left">
    <tr><td class="portletTitle" NOWRAP>worldclock title:${worldclocktitle}</td>
        <td CLASS="portletTitleControls"  NOWRAP>
            <a href="${worldclockview}">view</a>
            <a href="${worldclockedit}">edit</a>
            <a href="${worldclockmin}">minimize</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${worldclockcontent}
     </td>
    </tr>
  </table>

  <BR/>

<!--  This table is the bottom portlet in the first column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet bottom left">
    <tr>
    <td class="portletTitle" NOWRAP>test title:${testtitle}</td>
        <td CLASS="portletTitleControls" NOWRAP>
            <a href="${testview}">view</a>
            <a href="${testedit}">edit</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${testcontent}
     </td>
    </tr>
  </table>


  </TD>
```

```
  <TD VALIGN="top">


<!--  This table is the top portlet in the second column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet top right">
  <tr>
        <td class="portletTitle" NOWRAP>simple title:${simpletitle}</td>
        <td CLASS="portletTitleControls" NOWRAP>
            <a href="${simpleview}">view</a>
            <a href="${simpleedit}">edit</a>
            <a href="${simplemin}">minimize</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${simplecontent}
     </td>
    </tr>
  </table>

    </TD>
    </TR>
    </table>


</portlet:init>
```

You can include the following formatting to the previous example JSP file immediately once you have declared the tag library.

```
<STYLE TYPE="TEXT/CSS">
BODY {
    font-family:Verdana,sans-serif; font-size:70%
}
.portletTitle {
    text-align: left;border-top: #000000 1px solid; border-bottom: #000000 1px solid; FONT-SIZE: 60.0%;
    COLOR: #ffffff; FONT-FAMILY: Verdana, Arial, Helvetica, sans-serif; BACKGROUND-COLOR: #5495d5;
}
.portletTitleControls {
    text-align: right;border-top: #000000 1px solid; border-right: #000000 1px solid; border-bottom: #000000
    1px solid; FONT-SIZE: 60.0%; COLOR: #ffffff; FONT-FAMILY: Verdana, Arial, Helvetica, sans-serif;
    BACKGROUND-COLOR: #5495d5;
}
.portletTitleControls A {
    COLOR: #ffffff; text-decoration:none; border:#5495d5 1px solid;border-left:white 1px solid;
    padding-left:0.5em; padding-right:0.5em;
}
.portletTitleControls A:hover {
    COLOR: #ffffff; text-decoration:none; border-top:white 1px solid;
   border-bottom:white 1px solid;border-right:white 1px solid;
}
.minimizeControl {
    font-weight:bold; font-size:100%;
}
.portletTable {
    border-left: gray 1px solid;
    border-bottom: gray 1px solid;
    border-right: gray 1px solid;
}
.portletBody {
    font-family:Verdana,sans-serif; font-size:70%
}

</STYLE>
```

Portlet aggregation using JavaServer Pages

The aggregation tag library generates a portlet aggregation framework to address one or more portlets on one page If you write JavaServer Pages, you can aggregate multiple portlets on one page using the aggregation tag library. This tag library does not provide full featured portal aggregation implementation, but provides a good migration scenario if you already have aggregating servlets and JavaServer Pages and want to switch to portlets.

To allow the customer to create a simple portal aggregation, the aggregation tag library also provides the following features.

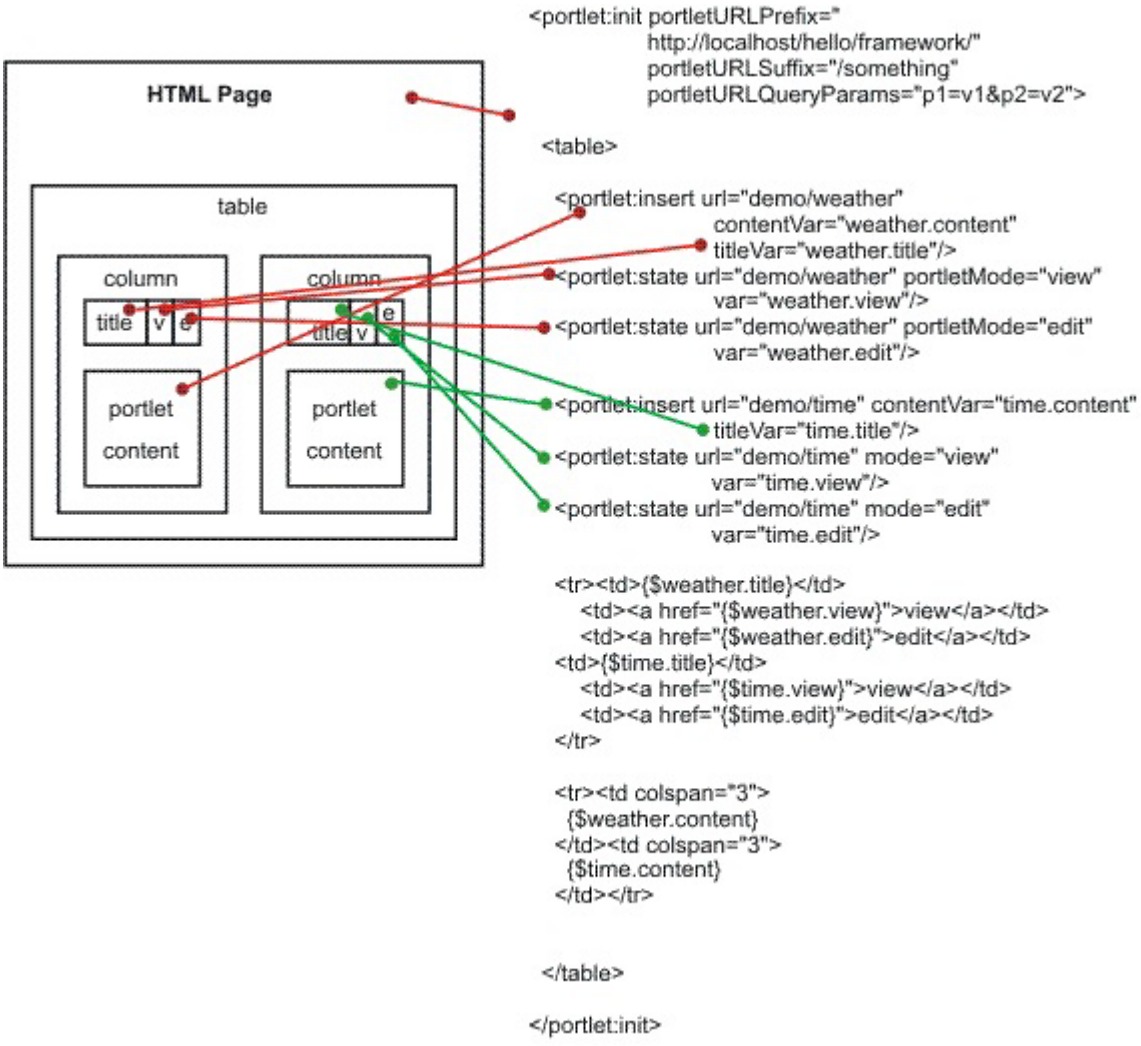
- Invoke a portlet's action method
- Render multiple portlets on one page
- Provide links to change the portlet's mode or window state

- Display the portlet's title
- Retain the portlet cookie state

The aggregation tag library and JavaServer Pages that use the aggregation tag library will only work with the WebSphere Application Server portlet container implementation because the protocol between the tags and the container is not standardized.

Refer to the Aggregation tag library attributes article for information on the aggregation tag library attributes.

The following diagram depicts how an HTML page would look like and what tags are used in order to create the page. Refer to the Aggregation tag library attributes article for information on the aggregation tag library attributes.



When you use the aggregation tag library, you must set the portletUrlPrefix attribute of the init tag to the aggregating application. You need to:

- Ensure that the portletUrlPrefix attribute is set to the following in the aggregator page.

  `"http://" + <server_address> + ":" + <server_port> + "/" + <aggregator context> + "/" <aggregator mapping>`
- Reference the aggregation JSP page within the `web.xml` file through a servlet mapping ending with `/*`. For example, `/aggregation/*`

When aggregating multiple portlets on a single page, special care must be used with the naming conventions of form attribute names in your portlets. Because your portlets are all on the same page, they all share the same HttpServletRequest. When one portlet is viewed the entire page is refreshed and form data is re-posted. Therefore, if there are multiple portlets that are aggregated on a single page with the same form attribute names, there could be logic corruption when form data is re-posted.

Aggregation tag library attributes

The aggregation tag library is used to aggregate multiple portlets on one page. This topic describes the attributes within the aggregation tag library.

Supported arguments include:

init

This tag initializes the portlet framework and has to be used in the beginning of the JSP. All other tags described in this section are only valid in the body of this tag, therefore the init tag usually encloses the whole body of a JSP. In case the current URL contains an action flag the action method of the corresponding portlet is called. The state and insert tags are sub-tags of the init tag.

The init tag has the following attributes:

- portletURLPrefix = ”<any string>”

  This URL defines the prefix used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet's render method, which is called by using the insert tag. This is a required attribute.
- portletURLSuffix = ”<any string>”

  This URL defines the suffix used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet's render method, which is called by using the insert tag. This is attribute optional.
- portletURLQueryParams = ”<any string>”

  This URL defines the query parameters used for PortletURLs. Portlet URLs are created either by the state tag or within a portlet's render method, which is called by using the insert tag. This is attribute optional.

scope, portlet

The scope tag and portlet tag are used to provide information that is necessary when a portlet application is installed under a multiple part context root, for example, /context1/context2. These tags also define which portlet windows should participate in portlet coordination via public render parameters, and add a render parameter to the newly created URL.

The urlParam tag has the following attributes:

- context = "/<context1>/<context2>"

  Specifies the context root of the portlet application in which the portlet is deployed. This attribute is required.
- portletname = "<portlet-name>"

  Specifies the portlet-name. This attribute is required.
- windowId = ”<any string>”

  Defines the window ID for the concrete portlet instance. This attribute is required.

The following is an example of how to use the scope and portlet tags:

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>

<portlet:scope>
    <portlet:portlet context="/myportletcontext1/myportletcontext2" portletname="MyPortlet" windowId="sample"/>
</portlet:scope>

<portlet:init portletURLPrefix="/myportalcontext/ ">
....
</portlet:init>
```

state

The state tag creates a URL pointing to the given portlet using the given state. You can place this URL either into a variable specified by the var attribute or you can write it directly to the output stream. This tag is useful to create URLs for HTML buttons, images, and other items such that when the URL is invoked, the state changes defined in the URL are applied to the given portlet.

The state tag has the following attributes:

- url = ”<context>/<portlet-name>”

  Identifies the portlet for this tag by using the context and portlet-name to address the portlet. This attribute is required.
- windowId = ”<any string>”

  Defines the window ID for the portlet URL created by this tag. This is attribute optional.
- var = ”<any string>”

  If defined the URL is written into a variable with the given scope and name, not to the output stream. This is attribute optional.
- scope = ”page|request|session|application”

  This attribute is only valid if the var attribute is specified. If defined, the URL is not written to the output stream but a variable is created in the given scope with the given name. The default is page. This is attribute optional.
- portletMode = ”view|help|edit|<custom>”

  This attribute sets the portlet mode.
- portletWindowState = ”maximized|minimized|normal|<custom>”

  This attribute sets the window state.
- action = ”true/false”

  This attribute defines whether this is an action URL. This is attribute optional. The default is false.

urlParam

Adds a render parameter to the newly created URL.

The urlParam tag has the following attributes:

- name = ”<any string>”

  Indicates the name of the parameter. This is attribute required.
- value = ”<any string>”

  Indicates the value of the parameter. This is attribute required.

insert

This tag calls the render method of the portlet and retrieves the content as well as the title. You can optionally place the content and title of the specified portlet into variables using the contentVar and titleVar attributes.

The insert tag has the following attributes:

- url = ”<context>/<portlet-name>” (mandatory) Identifies the portlet for this tag by using the context and portlet-name to address the portlet

  This is attribute required.
- windowId = ”<any string>”

  Defines the window ID of the portlet. This is attribute optional.
- contentVar = ”<any string>”

  If defined, the portlet's content is not written to the output stream but written into a variable with the given scope and name. This is attribute optional.
- contentScope = ”page|request|session|application”

This attribute is only valid if the contentVar tag is used. If defined, the portlet's content is written into a variable with the given scope and name, not to the output stream. The default is page. This is attribute optional.

- titleVar = ”<any string>”

  If defined the portlet's title is written into a variable with the given scope and name. If it is not defined, the title is ignored and not written to the output stream. This is attribute optional.

- titleScope = ”page|request|session|application”

  This attribute is only valid if titleVar tag is used. If defined, the portlet's title is written into a variable with the given scope and name, not to the output stream. The default is page. This is attribute optional.

Example: Using the portlet aggregation tag library

You can use the aggregation tag library to aggregate multiple portlets to have multiple and different content on one page. The library can be used by every JavaServer Pages (JSP) file that has been included by a servlet.

To use the portlet aggregation tag library, you must declare the tag-lib at the top of the JSP file using, `<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>`, as in the following example. The following JSP file example shows how to aggregate portlets on one page.

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>
<%@ page isELIgnored="false" import="java.util.Enumeration"%>

<portlet:init portletURLPrefix="/dummy/portletTagTest/" portletURLSuffix="/more" portletURLQueryParams="p1=v1&p2=v2">


   <portlet:insert url="worldclock/StdWorldClock" contentVar="worldclockcontent" titleVar="worldclocktitle"/>
    <portlet:state url="worldclock/StdWorldClock" portletMode="view" var="worldclockview"
       portletWindowState="maximized">
        <portlet:urlParam name="namea" value="valuea"/>
        <portlet:urlParam name="nameb" value="valueb"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" portletMode="edit" var="worldclockedit" portletWindowState="normal">
        <portlet:urlParam name="name1" value="value1"/>
        <portlet:urlParam name="name2" value="value2"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" portletMode="view" var="worldclockmin"
       portletWindowState="minimized">
        <portlet:urlParam name="namemin" value="valuemin"/>
        <portlet:urlParam name="namemin" value="valuemin"/>
    </portlet:state>

    <portlet:insert url="worldclock/StdWorldClock" windowId="min" contentVar="simplecontent" titleVar="simpletitle"/>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="view" var="simpleview"
  portletWindowState="maximized">
        <portlet:urlParam name="name3" value="value3"/>
        <portlet:urlParam name="name4" value="value4"/>
        <portlet:urlParam name="name5" value="value5"/>
        <portlet:urlParam name="name5" value="value5a"/>
        <portlet:urlParam name="name5" value="value5b"/>
        <portlet:urlParam name="name5" value="value5c"/>
    </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="edit" var="simpleedit"
  action="true" portletWindowState="normal">
        <portlet:urlParam name="name6" value="value6"/>
        <portlet:urlParam name="name6" value="value6z"/>
            </portlet:state>
    <portlet:state url="worldclock/StdWorldClock" windowId="min" portletMode="view" var="simplemin"
  portletWindowState="minimized">
        <portlet:urlParam name="name1" value="value1"/>
        <portlet:urlParam name="name2" value="value2"/>
    </portlet:state>

    <portlet:insert url="test/TestPortlet1" contentVar="testcontent" titleVar="testtitle"/>
    <portlet:state url="test/TestPortlet1" portletMode="view" var="testview" portletWindowState="maximized"/>
    <portlet:state url="test/TestPortlet1" portletMode="edit" var="testedit" portletWindowState="maximized"/>


<!--  This table is the outtermost table for creating two-column portal layout  -->
<TABLE border="0" CELLPADDING="3" CELLSPACING="8" WIDTH="100%">
<TR>
<TD VALIGN="top">

<!--  This table is the top portlet in the first column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet top left">
    <tr><td class="portletTitle" NOWRAP>worldclock title:${worldclocktitle}</td>
        <td CLASS="portletTitleControls"  NOWRAP>
```

```
            <a href="${worldclockview}">view</a>
            <a href="${worldclockedit}">edit</a>
            <a href="${worldclockmin}">minimize</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${worldclockcontent}
     </td>
    </tr>
  </table>

  <BR/>

<!--  This table is the bottom portlet in the first column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet bottom left">
    <tr>
    <td class="portletTitle" NOWRAP>test title:${testtitle}</td>
        <td CLASS="portletTitleControls" NOWRAP>
            <a href="${testview}">view</a>
            <a href="${testedit}">edit</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${testcontent}
     </td>
    </tr>
  </table>


  </TD>



  <TD VALIGN="top">


<!--  This table is the top portlet in the second column -->

  <table border="0" width="100%" CELLPADDING="3" CELLSPACING="0" CLASS="portletTable" SUMMARY="portlet top right">
  <tr>
        <td class="portletTitle" NOWRAP>simple title:${simpletitle}</td>
        <td CLASS="portletTitleControls" NOWRAP>
            <a href="${simpleview}">view</a>
            <a href="${simpleedit}">edit</a>
            <a href="${simplemin}">minimize</a>
        </td>
    </tr>
    <tr>
    <td CLASS="portletBody" COLSPAN="2">
      ${simplecontent}
     </td>
    </tr>
  </table>

    </TD>
    </TR>
    </table>


</portlet:init>
```

You can include the following formatting to the previous example JSP file immediately once you have declared the tag library.

```
<STYLE TYPE="TEXT/CSS">
BODY {
    font-family:Verdana,sans-serif; font-size:70%
}
.portletTitle {
    text-align: left;border-top: #000000 1px solid; border-bottom: #000000 1px solid; FONT-SIZE: 60.0%;
    COLOR: #ffffff; FONT-FAMILY: Verdana, Arial, Helvetica, sans-serif; BACKGROUND-COLOR: #5495d5;
}
.portletTitleControls {
    text-align: right;border-top: #000000 1px solid; border-right: #000000 1px solid; border-bottom: #000000
    1px solid; FONT-SIZE: 60.0%; COLOR: #ffffff; FONT-FAMILY: Verdana, Arial, Helvetica, sans-serif;
    BACKGROUND-COLOR: #5495d5;
}
.portletTitleControls A {
    COLOR: #ffffff; text-decoration:none; border:#5495d5 1px solid;border-left:white 1px solid;
    padding-left:0.5em; padding-right:0.5em;
}
.portletTitleControls A:hover {
    COLOR: #ffffff; text-decoration:none; border-top:white 1px solid;
   border-bottom:white 1px solid;border-right:white 1px solid;
}
.minimizeControl {
```

```
    font-weight:bold; font-size:100%;
}
.portletTable {
    border-left: gray 1px solid;
    border-bottom: gray 1px solid;
    border-right: gray 1px solid;
}
.portletBody {
    font-family:Verdana,sans-serif; font-size:70%
}

</STYLE>
```

Portlet preferences

Preferences are set by portlets to store customized information. By default, the PortletServingServlet servlet stores the portlet preferences for each portlet window in a cookie. However, you can change the location to store them in either a session, an `.xml` file, or a database.

Storing portlet preferences in cookies

The attributes of the cookie are defined as follows:

Path
context/portlet-name/portletwindow

Name:
The name of the cookie has the fixed value of **PortletPreferenceCookie**.

Value
The value of the cookie contains a list of preferences by mapping to the following structure:

*['/' pref-name *['=' pref-value]]

All preferences start with '/' followed by the name of the preference. If the preference has one or more values, the values follow the name separated by the '=' character. A null value is represented by the string '#*!0_NULL_0!*#'. As an example, the cookie value may look like, `/locations=raleigh=boeblingen/regions=nc=bw`

Customizing the portlet preferences storage

You can override how the cookie is handled to store preferences in a session, an `.xml` file or database. To customize the storage, you must create a filter, servlet or JavaServer Pages file as new entry point that wraps the request and response before calling the portlet. Examine the following example wrappers to understand how to change the behavior of the PortletServingServlet to store the preferences in a session instead of cookies.

The following is an example of how the main servlet manages the portlet invocation.

```
public class DispatchServlet extends HttpServlet
{
    ...
  public void service(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
  {
 response.setContentType("text/html");

 // create wrappers to change preference storage
 RequestProxy req = new RequestProxy(request);
 ResponseProxy resp = new ResponseProxy(request, response);

 // create url prefix to always return to this servlet
                ...
                req.setAttribute("com.ibm.wsspi.portlet.url.prefix", urlPrefix);

 // prepare portlet url
 String portletPath = request.getPathInfo();
```

```
                ...

 // include portlet using wrappers
 RequestDispatcher rd = getServletContext().getRequestDispatcher(modifiedPortletPath);
                rd.include(req, resp);
  }
}
```

In the following example, the request wrapper changes the cookie handling to retrieve the preferences out of the session.

```
public class RequestWrapper extends HttpServletRequestWrapper
{
    ...
    public Cookie[] getCookies() {
        Cookie[] cookies = (Cookie[]) session.getAttribute("SessionPreferences");
        return cookies;
    }
}
```

In the following example, the response wrapper changes the cookie handling to store the preferences in the session:

```
public class ResponseProxy extends HttpServletResponseWrapper
{
   ...
    public void addCookie(Cookie cookie) {
        Cookie[] oldCookies = (Cookie[]) session.getAttribute("SessionPreferences");
        int newPos = (oldCookies == null) ? 0 : oldCookies.length;
        Cookie[] newCookies = new Cookie[newPos+1];
        session.setAttribute("SessionPreferences", newCookies);

        if (oldCookies != null) {
        System.arraycopy(oldCookies, 0, newCookies, 0, oldCookies.length);
        }
        newCookies[newPos] = cookie;
    }
}
```

Portlet coordination

You can use either the events mechanism or the public render parameters mechanism to coordinate portlets within a portal.

JSR 286 defines these two mechanisms as follows:

- Events: Loose-coupling of portlets during action phase.
- Public render parameters: Sharing of view state between portlets.

The WebSphere Application Server portlet container supports both concepts with one exception. The wiring of events on a portal level is not supported.

To make use of the public render parameters on a portal page defined with the aggregation tag library, you must explicitly declare the portal scope:

```
<%@ taglib uri="http://ibm.com/portlet/aggregation" prefix="portlet" %>
<%@ page isELIgnored ="false"%>

<!-- define portal scope for public render parameters -->
<portlet:scope>
  <portlet:portlet context="/simpleportlet" portletname="SimplePortlet1" windowId="id1"/
  <portlet:portlet context="/simpleportlet2" portletname="SimplePortlet2" windowId="id2"/>
</portlet:scope>

<portlet:init portletURLPrefix="/my-webapp-context/my-portal-jsp-url-pattern/">
<!-- insert your portlets here, wrapped with your html markup -->
...
   <portlet:insert url="simpleportlet/SimplePortlet1" windowId="id1" titleVar="portlettitle_1"/>
```

```
...
   <portlet:insert url="simpleportlet2/SimplePortlet2" windowId="id2" titleVar="portlettitle_1"/>
...
</portlet:init>
```

The public render parameters are only visible to the portlet windows mentioned within this defined scope. This condition assumes, as a prerequisite, that the corresponding portlets have also declared support for a given public render parameter in their portlet.xml file, according to the JSR 286 specification:

```
...
   <portlet>
      <portlet-name>portlet_name1</portlet-name>
       ...
       <supported-public-render-parameter>foo</supported-public-render-parameter>
   </portlet>
   <portlet>
      <portlet-name>portlet_name2</portlet-name>
       ...
       <supported-public-render-parameter>foo</supported-public-render-parameter>
   </portlet>
   <public-render-parameter>
       <identifier>foo</identifier>
       <qname xmlns:x="http://example.com/params">x:foo2</qname>
   </public-render-parameter>
...
```

Converting portlet fragments to an HTML document

A portlet only delivers fragment output whereas a servlet typically delivers document output. However, you can use the PortletServingServlet servlet, which is similar to the FileServingServlet servlet, to address portlets like servlets.

About this task

A default document servlet filter, the DefaultFilter filter, is applied to the PortletServingServlet servlet to return the portlet‘s content inside of a document. This filter only applies to requests, not to includes or forwards using the RequestDispatcher method. A servlet filter that is used to embed the portlet‘s content into a document is called the document servlet filter. You can define additional document servlet filters in a `.xml` file. The FilterRequestHelper attribute within com.ibm.wsspi.portletcontainer.util is provided to assist the document servlet filters in analyzing a request regarding filter chain and portlet information. It is used in supporting dynamic portlet titles, as a marker for redirection for document servlet filters and to ensure that document conversion is completed once.

Procedure

1. Add a new document servlet filter. The filter capability is a server feature, therefore all filters must be installed into the server to use the filter capability of the server. The filters need to be available in any classes or library directory on a server level. You must also register the filter in a `plugin.xml` file within the root of a Java archive (JAR) file. The following is an example of how to register the filter in a `plugin.xml` file.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <?eclipse version="3.0"?>
   <plugin id="sample.plugin" name="Customer_Plugin" provider-name="Customer" version="1.0.0">
     <extension point="com.ibm.ws.portletcontainer.portlet-document-filter-config">
       <portlet-document-filter class-name="sample.filter.CustomFilter" order="200" />
     </extension>
   </plugin>
   ```

2. Optional: Set dynamic portlet titles by providing the dynamic title as a request attribute. The PortletServingServlet servlet supports dynamic portlet titles by providing the dynamic title as a request attribute, FilterRequestHelper.DYNAMIC_TITLE. This attribute returns the dynamic portlet title if it has been set by the portlet, otherwise it returns the static portlet title of the `portlet.xml` file if defined.

DYNAMIC_TITLE = 'javax.portlet.title' The DefaultFilter uses this request attribute to set the document title while converting the fragment into a document. If you want the filter to support browser caching or dynamic portlet titles, you must cache the complete portlet content.

3. Specify cache handling for the portlet rendering call to support dynamic title. Redirection for document servlet filters

   A document servlet filter can set a marker as request attribute, FilterRequestHelper.REDIRECT. This marker ensures that the portlet container returns to the document servlet filter after the portlet action has been called prior to any render calls. You must define the following constants, REDIRECT = 'com.ibm.websphere.portlet.action' and REDIRECT_VALUE = 'redirect'. The DefaultFilter uses this request attribute to provide special cache handling for the portlet rendering call to support dynamic title.

4. Convert the portlet's fragment into a valid document. Document conversion must be completed only once. Therefore each document servlet filter must ensure that the fragment has not yet been converted to a document previously. If the document servlet filter converts the fragment to a document, the request attribute FilterRequestHelper.DOCUMENT must be set to FilterRequestHelper.DOCUMENT_VALUE. This request attribute marks whether the conversion still needs to be completed. The following constants are defined, DOCUMENT = 'com.ibm.websphere.portlet.filter' and DOCUMENT_VALUE = 'document'. The DefaultFilter uses this request attribute to check whether it should convert the fragment to an Hypertext Markup Language (HTML) document. For example, this allows another document servlet filter in front to convert the fragment into a valid Wireless Markup Language (WML) document instead.

Assembling portlets

Portlet Uniform Resource Locator (URL) addressability

You can request a portlet directly through a Uniform Resource Locator (URL) to display its content without portal aggregation. The PortletServingServlet servlet registers each web application that contains portlets. It is similar to the FileServingServlet servlet of the web container that serves resources. The PortletServingServlet servlet supports direct rendering of portlets into a full browser page by a URL request.

You can invoke each portlet by its context root and name with the URL mapping /<portlet-name> that is created for each portlet. For example:

```
http://<host>:<port>/<context-root>/<portlet-name> For example,
 http://localhost:9080/portlets/TestPortlet1
```

The context root identifies the web application archive (WAR) file that contains the portlet. The portlet name uniquely identifies the portlet with a portlet application of a WAR file. The DefaultDocumentFilter servlet only supports HTML, UTF8 encoding and an extended URL form based on the basic URL form as shown above.

You can only display one portlet at a time using the PortletServingServlet servlet. If you want to aggregate multiple portlets on the page, you need to use the aggregation tag library. Refer to the Portlet aggregation using JavaServer Pages article for additional information.

Because a portlet only delivers fragment output whereas a servlet usually delivers document output, a mechanism is introduced to convert the fragment into a document, called the PortletDocumentFilter filter. By default, the PortletDocumentFilter filter only supports converting HTML. The conversion is implemented using a servlet filter before the PortletServingServlet servlet is initiated to return the portlet‘s content inside of a document. This default document servlet filter only applies to URL requests, not for includes or forwards using the RequestDispatcher method. You can create servlet filters to support other markups additional document servlet filters. Refer to the Converting portlet fragments to an HTML document article, for additional information.

The PortletServingServlet servlet does not persist portlet preferences in a XML file or database. It places the portlet preferences directly into a cookie to store the preferences persistently. Refer to the Portlet preferences article, for additional information on how to change this behavior.

Portlet URL syntax

You can add additional portal context such as portlet mode or window state. You can access the PortletServingServlet servlet by using a URL mapping that has the following structure:

```
http://host:port/context/portlet-name [/portletwindow[/ver [/action |
 /resource[/id=custom-id][/cacheability]] [/mode] [/state] [rparam][/?name]]]
```

Any differing URL structure results in a com.ibm.wsspi.portletcontainer.InvalidURLException exception. Empty strings are not permitted as parameter values and creates an InvalidURLException exception. The following is a list of valid parameters:

http:// host:port/context/*portlet-name*
: This is the minimum URL required to access a portlet. A default portlet window called “default” is created. The *portlet-name* variable is case-sensitive.

/*portletwindow*
: This parameter identifies the portlet window. You must set this parameter if you choose to add more portal context information to the URL.

/ver=*major.minor*
: This optional parameter is used to define the version of the portlet API that is used. You must set this parameter if you choose to add more portal context information to the URL. Only versions “1.0” and “2.0” are supported. Any other version creates an InvalidURLException exception.

/action
: This is a required parameter if you call the action method of the portlet. The action parameter causes the action process of the portlet to be called. After the action has been completed, a redirect is automatically issued to call the render process. To control the subsequent render process, a document servlet filter can set a request attribute with name “com.ibm.websphere.portlet.action” and value “redirect” to specify that the portlet serving servlet directly returns after action without calling the render process.

/mode=view | edit | help | *custom-mode*
: This optional parameter defines the portlet mode that is used to render the portlet. The default mode is “view”. The value is not case-sensitive. For example, “Vew”, “view” or “VIEW” results in the same mode.

/state=normal | maximized | minimized | *custom-state*
: This optional parameter defines the window state that is used to render the portlet. The default state is “normal”. The value is not case-sensitive, for example, “Normal”, “normal”, or “NORMAL” results in the same state.

*** [/rparam=***name* ***[=***value***]]**
: This optional parameter specifies render parameters for the portlet. Repeat this parameter chain to provide more than one render parameter. For example, /rparam=invitation/ rparam=days=Monday=Tuesday.

?name=value&name2=value2 ...
: Query parameters may follow optionally. They are not explicitly supported by the portlet container, but they do not invalidate the URL format.

/action | /resource
: This parameter defines the methods of the portlet that is called. Valid values are no, action or resource parameter. No specific method defined calls the render method. The resource parameter is only supported for JSR 286 portlets.

/resource [/id=custom-id] [/cacheability=cacheLevelFull | cacheLevelPortlet | cacheLevelPage]
Set this parameter to define the method of the portlet to be called. No redirection occurs. No other method of the portlet is called. To control the resource parameter, you can add an additional ID parameter to provide a resource serving identifier that is passed through to the portlet. The cacheability parameter defines the cache level of this resource URL. This parameter is only supported with JSR 286 portlets .

The following list includes examples of valid JSR 168 and JSR 286 URLs:

- http:// localhost:9080/sample/WorldClock
- http:// localhost:9080/sample/WorldClock/myPortlet/ver=1.0/mode=edit/rparam=timezone=UTC
- http:// localhost:9080/sample/WorldClock/myPortlet/ver=1.0/action/state=maximized?timezone=UTC
- http://localhost:9080/sample/WorldClock/myPortlet/ver=2.0/resource/id=somePicture.jpg

Example: Configuring the extended portlet deployment descriptor to disable PortletServingServlet

Portlet URL serving supports direct access to all functions and states of a portlet by creating the appropriate URLs. In a production setup where the portlet is served through an enterprise portal application that applies its own access control, is considered a security risk. By setting the portletServingEnabled property to false, an administrator can ensure that a sensitive portlet is never accessed by direct URL serving.

Extensions for the portlet deployment descriptor are defined within a file called `ibm-portlet-ext.xmi`. This deployment descriptor is an optional descriptor that you can use to configure WebSphere extensions for the portlet application and its portlets. For example, you can disable the PortletServingServlet servlet for the portlet application in the extended portlet deployment descriptor.

The `ibm-portlet-ext.xmi` extension file is loaded during application startup. If there are no extension files specified with this setting, the default values of the portlet container are used.

The default for the portletServingEnabled attribute is true. The following is an example of how to configure the application so that a PortletServingServlet servlet is not created for any portlet on the portlet application.

```
<?xml version="1.0" encoding="UTF-8"?>
<portletappext:PortletApplicationExtension xmi:version="1.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:portletappext="portletapplicationext.xmi"
    xmlns:portletapplication="portletapplication.xmi"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmi:id="PortletApp_ID_Ext"
    portletServingEnabled="false">
  <portletappext:portletApplication href="WEB-INF/portlet.xml#myPortletApp"/>
</portletappext:PortletApplicationExtension>
```

Chapter 19. Developing SCA composites

This page provides a starting point for finding information about Service Component Architecture (SCA) composites, which consist of components that implement business functions in the form of services.

You typically do not deploy SCA composites directly onto a product server. To deploy SCA composites, you import SCA composites as assets to the product repository and add the assets to business-level applications.

Selecting the implementation type for an SCA composite

A benefit of Service Component Architecture (SCA) is that you can use existing assets in an application, without having to change the asset implementation. Many enterprises have application assets developed using a variety of technologies and frameworks, including assets previously deployed in a WebSphere Application Server environment, which is predominantly Java Platform, Enterprise Edition (Java EE) or Java-centric. SCA supports several service implementation technologies.

Before you begin

Determine which application files or artifacts (assets) you want to deploy in an SCA application. If necessary, develop the files or artifacts.

About this task

You can use SCA composites to:

- Reuse existing service implementations in the definition of an SCA component for use in an SCA composite application.
- Enable the services within those technologies to be exposed or use a service over the supported access bindings without changing the business logic of the service.

Each component within an SCA assembly specifies the implementation technologies which provide the specific business logic which implements the services described in the component definition. A composite is composed of 1-to-many components, each of which can be made up of differing implementation technologies.

SCA supports the following implementation programming model technologies. The product supports all of these implementation types on OSOA but does not support the implementation types OSGi, Java EE (JEE), Spring, or Widget on OASIS.

implementation.java
The implementation conforms to the SCA Java Component Implementation specification.

implementation.composite
The implementation conforms to an SCA composite as defined in the SCA Assembly Model specification.

implementation.web
The implementation conforms to Java EE 1.5 for Web modules and Web archive (WAR) files.

implementation.ejb
The implementation conforms to Java EE 1.5 for Enterprise JavaBeans (EJB) 2.1 and 3.0 Stateless Session Beans.

implementation.jee
The implementation conforms to Java EE 1.5 for enterprise archive (EAR) files.

implementation.osgiapp
The implementation conforms to the OSGi application programming model as supplied in the product.

implementation.spring
The implementation conforms to the SCA Spring Component Implementation specification.

WebSphere Application Server does not include the Spring 2.5.5 package. You must obtain the Spring 2.5.5 package from the supplying vendor.

implementation.widget
The implementation is SCA-enhanced JavaScript contained within an HTML page. Unlike the other implementation types, this component runs on the server but it is returned to browsers from the server. You wire services that JavaScript code references through SCA bindings.

Procedure

1. Select the implementation type to use for the SCA component.

Table 84. Supported SCA implementation types. Based on the implementation technology of an asset, select the SCA implementation type to use in the SCA component.

Asset implementation technology	SCA implementation type
SCA Java	`implementation.java`
SCA assembly	`implementation.composite`
Java EE application (EAR)	`implementation.jee`
Java EE EJB module, session bean, or message-driven bean	`implementation.ejb`
Java EE Web module in a Java EE application	`implementation.web`
Enterprise bundle archive (EBA) artifact that uses OSGi bundles and Blueprint components	`implementation.osgiapp`
Bean that follows the Java 2 Platform, Standard Edition (J2SE) programming model in a Spring 2.5.5 container.	`implementation.spring`
HTML page enriched with JavaScript code that contains SCA for return to a browser. An SCA service returns data in JavaScript. The data can be in Atom collections or in JavaScript Object Notation (JSON) format.	`implementation.widget`

2. Configure the binding for the selected implementation type and use the binding in an SCA component or application.

   See topics in the related links for information on specifying implementations and bindings.

Results

The SCA composite makes the appropriate environmental transitions to connect the implementation technology to any exposed bindings that are also in the composite definition.

What to do next

Deploy the SCA composite or application in a business-level application.

Developing Service Component Architecture (SCA) services

To develop SCA service implementations, you can use either a top-down development approach starting with an existing Web Services Description Language (WSDL) file or you can use a bottom-up development approach starting from an existing Java interface or implementation. When using either the top-down or bottom-up development methodologies, you can develop service clients or use tools to map business exceptions on remotable interfaces.

Developing SCA services from existing WSDL files

You can develop a Service Component Architecture (SCA) service implementation when starting with an existing Web Services Description Language (WSDL) file.

Before you begin

Locate the WSDL file that defines the SCA service that you want to implement. You can develop a WSDL file or obtain one from an existing SCA service. The WSDL file describes your service interface as a WSDL portType and includes XSD schema definitions of your business data.

The product supports Web Services Description Language (WSDL) Version 1.1 definitions that also conform to the WS-I Basic Profile Version 1.1 and Simple SOAP Binding Profile 1.0 standards, and use the document literal style. All these conditions are required for support.

About this task

There are two ways to develop an SCA service implementation:

- Top-down development starting with an existing Web Services Description Language (WSDL)
- Bottom-up development starting from existing Java code that uses Java Architecture for XML Binding (JAXB) data types

The top-down development approach takes advantage of the interoperable XML-based WSDL, and XSD interface and data definitions.

This task describes the steps when using the top-down development approach to develop an SCA service implementation in Java when starting from a WSDL interface and XSD data definitions.

Note: It is a best practice to use the top-down methodology to develop SCA service implementations because this approach uses the capabilities of the XML interface description and provides greater ease in interoperability across platforms, bindings, and programming languages.

Use the **wsimport** command-line tool to generate the Java representations of your business service interfaces and your business data when an existing WSDL file describes the wanted SCA service interface as a WSDL portType, along with XSD schema definitions of your business data. The **wsimport** tool generates Java classes that you can use to write a Java implementation that reflects your business logic. The result is a Plain Old Java Object (POJO) implementation of the generated interface using the generated JAXB data types. By adding the @Service annotation to the Java implementation, the annotation defines the Java implementation as an SCA service implementation.

The generated annotated Java classes that correspond to your business data contain all the necessary information that the JAXB runtime environment requires to build and parse the XML for marshaling and unmarshaling. In other words, the data programming model is limited to object instantiation and the use of getter and setter methods, and you do not need to write code to convert the data between the XML wire format and the Java application.

Note: The product uses XML marshaling as defined by JAXB to marshal and unmarshal data across a remotable interface. If you start with a remotable Java interface for your implementation rather than starting with a WSDL portType interface, be careful when selecting the input and output Java data types and ensure that you understand which data is preserved across JAXB marshaling and unmarshalling. However, when authoring an implementation on a local interface, you can use any Java type because local interfaces use pass-by-reference semantics, which implies no data is copied.

Note: The product does not support using a WSDL file when the Java mapping requires holder classes. The product uses the JAX-WS specification to define the mapping between WSDL files and Java,

including the mapping between a WSDL portType object and a Java interface. When you have WSDL portType objects with operations that use in-out parameters or operations that use multiple output parameters, the JAX-WS specification uses instances of the javax.xml.ws.Holder class in the mapping of the WSDL portType object to a Java interface. When using the product, do not use a WSDL file when the Java mapping requires holder classes. Instead, use a WSDL file that does not map to holder classes.

When you develop an SCA service when starting from an existing WSDL file, the interface is considered a remotable interface. The remotable interface uses pass-by-value semantics, which implies your data is copied.

You can use and deploy the resulting Java implementation as an SCA component that is defined in a composite definition. The composite definition defines SCA artifacts, such as service references, imports, and exports. The component is defined in terms of development artifacts such as the WSDL, the Java implementation, and bindings that are defined during deployment.

The JAXB APIs require that you register Java class types that you want to marshal or unmarshall with a JAXBContext class. The product runtime environment registers these Java class types for you by introspecting your Java interface. When this introspection occurs, be careful of possible problems when polymorphism (inheritance) is used. When an interface is defined in terms of a base superclass type, and you want to pass argument instances of a derived subclass type at run time, the subclasses are not known to the JAXBContext class by simply introspecting the interface parameter types.

In JAXB, you can use the javax.xml.bind.annotation.XmlSeeAlso annotation to solve this problem with polymorphism. Place the @XmlSeeAlso annotation on the generated Java interface that is generated with the @WebService annotation, to refer to additional JAXB derived subclass classes that are added to the JAXBContext class along with those classes introspected from the interface parameters.

Procedure

1. Use the **wsimport** command-line tool to develop SCA Java representations of your business service interfaces and your business data.

   The **wsimport** tool processes a WSDL file and generates Java classes and the JAXB data types that are used to create the SCA service.

   It is important to include all generated classes within your application archive, including the classes that you might not directly reference in your Java implementation. Even if you have simple interfaces that pass simple parameter types like String and Integer, or where no JAXB data types are necessary, be sure to include all classes, including indirect references, in this code generation step.

   - Run the wsimport command to generate the artifacts.

   The **wsimport** tool is located in the *app_server_root*/bin/ directory.

   The **-keep** option specifies to keep the generated Java source files and the compiled class files.
2. Locate the Java interface that directly corresponds to your WSDL portType from the generated artifacts. The interface is generated with an @WebService annotation, and it is an interface and not a class file.
3. Complete the implementation of your SCA service. Write a Java implementation of the generated Java interface that reflects your business logic. The Java implementation is a Plain Old Java Object (POJO) implementation of the generated interface using the generated JAXB data types. This implementation is annotated based on the SCA Java component implementation programming model.
4. Annotate the Java implementation. Add the @Service annotation to the Java implementation to specify this implementation is an SCA service. When you complete this step, you have created an SCA component implementation.
5. Define a component within a composite definition using this component implementation. In the definition of your composite, define a component that refers to the original WSDL portType interface and the SCA implementation.

a. Under the <component> element, create a <implementation.java> child element that refers to the class name of your POJO component implementation.
b. Under the <component> element, create a <service> child element.
c. Under the <service> element, create a <interface.wsdl ..> element that refers to the WSDL portType. The @name attribute of the <service> element must match the unqualified class name of your Java interface.

You now have a component with a well-defined component name and service name with a well-defined interface.

In addition to these aspects of your component definition described by these development procedures, there are other aspects of defining a component. These aspects include adding bindings, configuring property values, defining intents, attaching policy sets, and resolving references. You can create multiple components using this same implementation, but all component definitions are the same with respect to the <implementation.java>, <interface.wsdl> and <service> elements described in this step.

6. Deploy the SCA service by creating the SCA business level application from a deployable composite.

   In the previous step, you defined a component providing your SCA service within a composite definition. This composite is either a deployable composite, or one that is used recursively as a composite implementation of a component in a higher-level composite. To learn how to deploy the SCA service, read about deploying and administering business-level applications.

Results

You have created an SCA implementation by starting with an existing WSDL file.

Example

The following example illustrates using an existing WSDL interface to generate a Java interface that is used to create a Java implementation that is an SCA service.

1. Copy the following sample account.wsdl WSDL file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:account="http://www.myaccount.com/account"
     targetNamespace="http://www.myaccount.com/account"
     name="AccountService">

      <wsdl:types>
         <schema targetNamespace="http://www.myaccount.com/account"
             xmlns="http://www.w3.org/2001/XMLSchema"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:account="http://www.myaccount.com/account">

               <element name="computeAccountAverage">
                 <complexType>
                     <sequence>
                         <element name="account" type="account:Account" />
                         <element name="days" type="xsd:int" />
                     </sequence>
                 </complexType>
             </element>
             <element name="computeAccountAverageResponse">
                 <complexType>
                     <sequence>
                         <element name="return" type="xsd:float" />
                     </sequence>
                 </complexType>
             </element>

             <complexType name="Account">
                 <attribute name="accountNumber" type="xsd:int" />
                 <attribute name="accountID" type="xsd:string" />
```

```
                <attribute name="accountType" type="xsd:string" />
                <attribute name="balance" type="xsd:float" />
            </complexType>

        </schema>
    </wsdl:types>

    <wsdl:message name="computeAccountAverageRequest">
        <wsdl:part element="account:computeAccountAverage"
            name="parameters" />
    </wsdl:message>

    <wsdl:message name="computeAccountAverageResponse">
        <wsdl:part element="account:computeAccountAverageResponse"
            name="parameters" />
    </wsdl:message>

    <wsdl:portType name="AccountService">
        <wsdl:operation name="computeAccountAverage">
            <wsdl:input message="account:computeAccountAverageRequest" name="accountReq"/>
            <wsdl:output message="account:computeAccountAverageResponse" name="accountResp"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="AccountServiceSOAP" type="account:AccountService">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="computeAccountAverage">
           <soap:operation
                soapAction="computeAccountAverage" />
            <wsdl:input name="accountReq">
               <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="accountResp">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>

     </wsdl:binding>

    <wsdl:service name="AccountWSDLService">
        <wsdl:port binding="account:AccountServiceSOAP"
            name="AccountServicePort">
            <soap:address location=""/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

2. Run the **wsimport** command from the *app_server_root*/bin/ directory.After generating the template files using the **wsimport** command, the following Java files are generated:

```
com/myaccount/account/Account.java
com/myaccount/account/AccountService.java
com/myaccount/account/AccountWSDLService.java
com/myaccount/account/ComputeAccountAverage.java
com/myaccount/account/ComputeAccountAverageResponse.java
com/myaccount/account/ObjectFactory.java
com/myaccount/account/package-info.java
```

3. Identify the generated Java interface from the generated classes.

```
//
// Generated By:JAX-WS RI IBM 2.1.1 in JDK 6 (JAXB RI IBM JAXB 2.1.3 in JDK 1.6)
//
package com.myaccount.account;
...
@WebService(name = "AccountService", targetNamespace = "http://www.myaccount.com/account")
...
public interface AccountService {
    /**
     *
     * @param days
     * @param account
     * @return
     *     returns float
     */
```

```
    @WebMethod(action = "computeAccountAverage")
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "computeAccountAverage", targetNamespace = "http://www.myaccount.com/account",
            className = "com.myaccount.account.ComputeAccountAverage")

    @ResponseWrapper(localName = "computeAccountAverageResponse", targetNamespace = "http://www.myaccount.com/account",
            className = "com.myaccount.account.ComputeAccountAverageResponse")
    public float computeAccountAverage(
        @WebParam(name = "account", targetNamespace = "")
        Account account,
        @WebParam(name = "days", targetNamespace = "")
        int days);

}
```

This code example is a Java interface, not merely a Java class. The @WebService annotation is present in this Java interface. It is important to know that this example is not the same as the generated @WebServiceClient class, com.myaccount.account.AccountWSDLService, which is not an interface and is not needed in your SCA application.

4. Complete the implementation of your SCA service by writing a Java implementation of this generated Java interface. Be sure to add the SCA @Service annotation to the implementation.

```
package com.myaccount.account;
import org.osoa.sca.annotations.Service;
@Service(AccountService.class)
public class AccountServiceImpl implements AccountService
    public float computeAccountAverage( Account account, int days) {

        // Write your business logic here.  Account is a
        // generated JAXB type and so use the JAXB programming model.
        // For example, object instantation is performed using
        // the ObjectFactory.createAccount()) method.
    }
}
```

By completing this step, you have completed a component implementation. Not only is this component implementation a Java implementation of a Java interface, but the @Service annotation signifies that this is a Java component implementation of an SCA service interface. The implementation class itself does not need all the JAX-WS or JAXB annotations. The runtime environment loads the appropriate annotations from the generated classes that the implementation refers to.

5. Create a component using the component implementation. You create a component definition in a composite that references the original WSDL portType interface and the SCA implementation. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. Shown here are the aspects of component creation that are common for all component definitions using the implementation developed in this example. This example also includes bindings that you can modify or omit for other components using this component implementation.

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
     targetNamespace="http://account.customer"
         name="accountComposite">
     <component name="BankingComponent">
         <implementation.java class="com.myaccount.account.AccountServiceImpl"/>

         <!-- The @name value matches the contents of the @Service which in turn
              comes from the WSDL portType. -->

         <service name="AccountService">

              <!-- This statement specifies the QName of the WSDL portType,
                   "http://www.myaccount.com/account#AccountService" in the syntax as
                   illustrated in the interface.wsdl statement. -->

             <interface.wsdl interface="http://www.myaccount.com/account#wsdl.interface(AccountService)" />
             <binding.ws/>

  <!-- This example uses the SCA web services binding.  However, it does not matter which SCA binding
       you choose.  -->
```

```
        </service>
    </component>
</composite>
```

6. After your component is defined as part of a deployable composite, either directly or recursively through use of one or more layers of components with composite implementation, you are ready to deploy the SCA service by creating an SCA business level application.

Developing SCA services with existing Java code

You can develop an Service Component Architecture (SCA) service implementation when starting with an existing Java application.

About this task

There are two ways to develop an SCA service implementation:

- Top-down development starting with an existing Web Services Description Language (WSDL)
- Bottom-up development starting from existing Java code that uses Java Architecture for XML Binding (JAXB) data types

The bottom-up development approach provides a simplified way to begin developing SCA services for the Java developer that does not desire to work with WSDL or XML schema (XSD) authoring or when building new SCA services that expose existing legacy implementations with Java interfaces.

The top-down development approach takes advantage of the interoperable XML-based WSDL, and XSD interface and data definitions.

This task describes the steps when using the bottom-up development approach to develop an SCA service implementation when starting with Java.

When using the bottom-up development methodology, begin by writing a Java interface and implementation that describes the desired business logic. This implementation is then packaged into an application archive file such as a web application archive (WAR) file or a Java archive (JAR) file that is subsequently is used by an SCA component that is configured with deployment information containing the bindings when the SCA application is deployed.

Note: It is a best practice to use the top-down methodology to develop SCA service implementations because this approach leverages the capabilities of the XML interface description and provides a greater ease in interoperability across platforms, bindings, and programming languages. To learn more about using the top-down methodology, read about developing SCA services from existing WSDL files.

Note: The product uses XML marshalling as defined by JAXB to marshal and unmarshal data across a remotable interface. If you start with a remotable Java interface for your implementation rather than starting with a WSDL portType interface, be careful when selecting the input and output Java data types and ensure you understand which data is preserved across JAXB marshalling and unmarshalling. However, when authoring an implementation on a local interface, you can use any Java type because local interfaces use pass-by-reference semantics, which implies no data is copied.

Note: The data marshalling and unmarshalling that is used to instantiate the copying of data over remotable interfaces is defined by the JAXB specification rather than by Java serialization or the java.io.Serializable or java.io.Externalizable interfaces. Because of this behavior, certain existing Java types are not suitable for use on remotable interfaces, as these types are not serialized using Java serialization. For data types that are not annotated, the class is introspected and its Java properties determine the data that is preserved in the copy. For data types that take advantage of JAXB annotations, you can customize the mapping of Java classes to XSD types and of Java

instances to XML documents. Custom Java serialization routines such as the readObject() or writeObject() are not applicable in this scenario. The SCA runtime environment takes an XML centric view of the business data and leverages the JAXB standards to define the mappings between the Java programming model and the XML data format on the wire.

Procedure

1. Access the existing Java interface that you want to expose as an SCA service.
2. Determine if you are using a local or a remotable interface.
   - If you are using a remotable interface, add the @Remotable annotation to the Java interface. The input and output Java data types on the remotable interface use pass-by-value semantics which implies your data is copied using XML serialization as defined by JAXB.
3. Complete the implementation of your SCA service. Write a Java implementation of the generated Java interface that reflects your business logic. The Java implementation is a Plain Old Java Object (POJO) implementation of the original interface.
4. Annotate the Java implementation. Add the @Service annotation to the Java implementation to specify this is an SCA service. When you complete this step, you have created an SCA component implementation.
5. Define a component within a composite definition using this component implementation. In the deinition of your composite, define a component that refers back to the original Java interface and the SCA implementation.
   a. Under the `<component>` element, create a `<implementation.java>` child element that refers to the class name of your POJO component implementation.
   b. Under the `<component>` element, create a `<service>` child element.
   c. Under the `<service>` element, create a `<interface.java ..>` element that refers back to the original Java interface. The @name attribute of the `<service>` element must match the unqualified class name of your Java interface.

   You now have a component with a well-defined component name and service name with a well-defined interface.

   In addition to these aspects of your component definition described by these development procedures, there are other aspects of defining a component. These aspects include adding bindings, configuring property values, defining intents, attaching policy sets, and resolving references. You can create multiple components using this same implementation, but all component definitions are the same with respect to the `<implementation.java>`, `<imterface.java>` and `<service>` elements described in this step.
6. Deploy the SCA service by creating the SCA business level application from a deployable composite.

   In the previous step, you defined a component providing your SCA service within a composite definition. This composite is either a deployable composite, or one that is used recursively as a composite implementation of a component in a higher-level composite. To learn how to deploy the SCA service, read about deploying and administering business-level applications.

Results

You have developed an SCA service using the bottom-up methodology by starting with an existing Java interface or implementation.

Example

The following example illustrates how to create a component implementation of a remotable SCA service interface starting from existing Java code:

1. Start with Java interface `myintf.NameGetter` using type `mypkg.Person`.

```
//NameGetter.java
package myintf;
import mypkg.Person;
```

```
public interface NameGetter {
    public String getName(Person p);
}

//Person.java

package mypkg;

public class Person {

    protected String firstName;
    protected String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String value) {
        this.firstName = value;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String value) {
        this.lastName = value;
    }
}
```

In this example, the mypkg.Person class is well-suited for use over a remotable interface, because it follows the JavaBeans pattern and contains a public getter and setter pair for its important data fields. The XML wire format used by the runtime environment will serialize and deserialize this class. However, other existing Java types that do not adhere to the JavaBeans pattern can cause problems as they dol not serialize correctly and data loss occurs. For this reason, it is a best practice to use a top-down development approach, starting from schema definitions and generating JAXB classes for use in the application programming model. See the developing SCA services from existing WSDL files to learn more about the top-down development approach.

2. Because we are creating a service with a remotable interface, add the @Remotable annotation.

```
//NameGetter.java
package myintf;

import mypkg.Person;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface NameGetter {
    public String getName(Person p);
}
```

3. Unless you have an existing Java implementation, write a Java implementation of the generated Java interface that reflects your business logic.

```
package myintf;
import mypkg.Person;

public class NameGetterImpl implements NameGetter {

    public String getName(Person p) {
        // Example "business logic"
        return p.getFirstName() + " " + p.getLastName();
    }

}
```

4. Add the @Service annotation to the Java implementation.

```
package myintf;
import mypkg.Person;
import org.osoa.sca.annotations.Service;

@Service(NameGetter.class)
public class NameGetterImpl implements NameGetter {

    public String getName(Person p) {
        // Example "business logic"
        return p.getFirstName() + " " + p.getLastName();
    }

}
```

5. Create a component using the component implementation. You will create a component definition in a composite that references the original Java implementation class, as well as its Java interface. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. Shown here are the aspects of component creation that are common for all component definitions using the implementation developed in this example. This example also includes bindings that you can modify or omit for other components using this component implementation.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://org.services.naming"
        name="NameServices">

      <component name="NamingServicesComponent">
         <implementation.java class="myintf.NameGetterImpl"/>
         <service name="NameGetter">

             <!-- The interface.java is not required because the run time can introspect it. -->

             <interface.java interface="myintf.NameGetter"/>

             <!-- The choice of bindings is not important for the example.  Here, both the SCA default and
                  web services bindings are configured. -->
             <binding.ws/>
             <binding.sca/>
         </service>
     </component>

</composite>
```

6. After your component is defined as part of a deployable composite, either directly or recursively through use of one or more layers of components with composite implementation, you are ready to deploy the SCA service by creating an SCA business level application.

Developing SCA service clients

You can develop a Service Component Architecture (SCA) service client starting with either a Java interface or a WSDL file for the SCA service that you want to invoke.

About this task

You can develop SCA service clients that can both access and invoke an SCA service that are based on the Service Component Architecture specification. An SCA client can consume a diverse set of services such as enterprise beans, web services, and other SCA services, through the capabilities of the respective SCA bindings and by using the Plain Old Java Object (POJO) client programming model.

To develop SCA service clients, you can start with either an existing Web Services Description Language (WSDL) file and use the **wsimport** tool to generate the Java interface or you can start with an existing Java interface.

Developing SCA client components starting with an existing WSDL file

When you have an existing WSDL file that describes your SCA service interface as a WSDL portType, along with XSD schema definitions of your business data, you can use the **wsimport** tool to generate the SCA Java representations of your business service interfaces and your business data. The **wsimport** tool generates Java classes that you can use to write a Java implementation that reflects your business logic. You can use the generated output of the proxy class and the JAXB data binding types in your Java client to invoke the SCA service using the simple POJO programming model.

The generated annotated Java classes that correspond to your business data contain all the necessary information that the Java Architecture for XML Binding (JAXB) runtime environment requires to build and parse the XML for marshaling and unmarshaling. In other words, the data programming model is limited to object instantiation and the use of getter and setter methods, and you do not need to write code to convert the data between the XML wire format and the Java application.

Now that you have the generated annotated Java classes, you must use the Java interface and data type classes to create the reference proxy as described in the developing SCA clients starting with a Java interface section.

Developing SCA client components starting with a Java interface

When you have a Java interface to your SCA service, obtained either by starting from a WSDL and generating the Java classes or by starting with Java code, use the Java interface and data type classes to create the reference proxy. If your client is designed so that its reference proxy is injected from the SCA container, the Java interface is the same type as your proxy field and this file contains the corresponding @Reference annotation. You can only create the static reference from another SCA component implementation that acts as a client of the original service. If your reference proxy is created programmatically, you must create a proxy variable that has the same type as your Java interface, and use an API such as `CompositeContext.getService(Class <B>interfaze, ...)` to create the reference proxy. The generated Java interface type is the interface parameter that is passed to this API. Read about locating and invoking SCA services to learn more about creating the reference proxy dynamically.

Regardless of whether the proxy is created by injection methods or programmatically, the Java interface is the class of the proxy and the generated JAXB types are the parameter types which includes inputs, outputs, and exceptions.

Considerations for local and remotable interfaces

It is important to understand that a remotable interface uses an XML wire format for data. Therefore, clients must use a JAXB-based programming model for the data types. In contrast, a local interface uses pass-by-reference semantics, so there is no data copy. Using the local interface, data is read and written without any special programming model such as JAXB.

Though WSDL-based interfaces are always remotable, you can also mark a Java interface that is not generated from a WSDL file as remotable by annotating it with the @Remotable annotation. The @Remotable annotation results in a data copy with XML serialization as defined by JAXB.

Defining the remotable interface is straightforward when you start with a WSDL interface, because you use the **wsimport** tool to generate the JAXB data types that you use when you write your SCA client. The remotable interface is less apparent when starting from a remotable Java interface, unless the Java types are decorated with JAXB annotations. XML serialization behaves differently than Java serialization. For an POJO that is not annotated, Java serialization preserves instance data including private fields, whereas JAXB serialization preserves JavaBeans properties.

The focus of this topic is the use of remotable interfaces.

You can develop a component that consumes or acts as a client of the target service using a component reference. In addition to consuming a service from another component's reference, the product also provides a mechanism for consuming an SCA service over the default binding from a non-SCA component.

Procedure

1. Determine if you are developing the SCA service client starting with an existing WSDL file or with an existing Java interface.
2. Develop the client Java interfaces and data types from a WSDL file if you are not starting with an existing Java interface. Use the **wsimport** command to generate the SCA service client Java interfaces.
3. Create the reference proxy based on the Java interface.
   a. Create a reference proxy field or setter method that has the same type as the generated Java interface
   b. Annotate this field or setter with the @Reference annotation.

   Now you have completed the steps required to add the reference to your Java component implementation
4. Create a component definition using the Java implementation.

   In the composite definition, add a <reference> element that refers back to the original interface and the field or setter of your SCA implementation. The reference is added as a child element of your component. The component is part of a composite definition.

   The <reference> name attribute must correspond to the field or setter that contains the @Reference annotation. For a field that contains the @Reference annotation, the name attribute must match exactly. For a setter that contains the @Reference annotation, use the usual Java conventions for translating an annotated setter into a corresponding field, which in turn must match the name attribute.

   For the interface:

   - If your SCA client development started with an existing WSDL file, create an <interface.wsdl> element as a child element of the <reference> element that points to the WSDL portType.
   - If your SCA client development started from existing Java interface, create an <interface.java> element as a child element of the <reference> element that points to the original Java interface. This is optional, since the runtime environment can introspect the Java interface.

   In addition to these aspects of your component definition described by these development procedures, there are other aspects of defining a component. These aspects include adding bindings, configuring property values, defining intents, attaching policy sets, and resolving references. You can create multiple components using this same implementation, but all component definitions are the same with respect to the <implementation.wsdl> element and <reference> element described in this step.
5. Deploy the SCA component by creating the SCA business level application from a deployable composite.

   In the previous step, you defined a component providing your SCA service within a composite definition. This composite is either a deployable composite, or one that is used recursively as a composite implementation of a component in a higher-level composite. To learn how to deploy the SCA service, read about deploying and administering business-level applications.

Results

You have created an SCA component that can consume an existing SCA service using a WSDL or Java interface.

Example

The following example illustrates using an existing WSDL interface to generate a Java interface that is used to create a Java implementation that is an SCA client. If you are starting with an existing Java interface, begin with step 4 to follow this example.

1. Copy the following sample account.wsdl WSDL file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:account="http://www.myaccount.com/account"
```

```
targetNamespace="http://www.myaccount.com/account"
name="AccountService">

 <wsdl:types>
    <schema targetNamespace="http://www.myaccount.com/account"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:account="http://www.myaccount.com/account">

          <element name="computeAccountAverage">
            <complexType>
                <sequence>
                    <element name="account" type="account:Account" />
                    <element name="days" type="xsd:int" />
                </sequence>
            </complexType>
        </element>
        <element name="computeAccountAverageResponse">
            <complexType>
                <sequence>
                    <element name="return" type="xsd:float" />
                </sequence>
            </complexType>
        </element>

        <complexType name="Account">
            <attribute name="accountNumber" type="xsd:int" />
            <attribute name="accountID" type="xsd:string" />
            <attribute name="accountType" type="xsd:string" />
            <attribute name="balance" type="xsd:float" />
        </complexType>

     </schema>
</wsdl:types>

<wsdl:message name="computeAccountAverageRequest">
    <wsdl:part element="account:computeAccountAverage"
        name="parameters" />
</wsdl:message>

<wsdl:message name="computeAccountAverageResponse">
    <wsdl:part element="account:computeAccountAverageResponse"
        name="parameters" />
</wsdl:message>

<wsdl:portType name="AccountService">
    <wsdl:operation name="computeAccountAverage">
        <wsdl:input message="account:computeAccountAverageRequest" name="accountReq"/>
        <wsdl:output message="account:computeAccountAverageResponse" name="accountResp"/>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="AccountServiceSOAP" type="account:AccountService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="computeAccountAverage">
       <soap:operation
            soapAction="computeAccountAverage" />
        <wsdl:input name="accountReq">
           <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output name="accountResp">
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>

 </wsdl:binding>

<wsdl:service name="AccountWSDLService">
    <wsdl:port binding="account:AccountServiceSOAP"
        name="AccountServicePort">
```

```
            <soap:address location=""/>
          </wsdl:port>
        </wsdl:service>
    </wsdl:definitions>
```

2. Run the **wsimport** command from the *app_server_root*/bin/ directory.After generating the template files using the **wsimport** command, the following Java files are generated:

```
com/myaccount/account/Account.java
com/myaccount/account/AccountService.java
com/myaccount/account/AccountWSDLService.java
com/myaccount/account/ComputeAccountAverage.java
com/myaccount/account/ComputeAccountAverageResponse.java
com/myaccount/account/ObjectFactory.java
com/myaccount/account/package-info.java
```

3. Identify the generated Java interface from the generated classes.

```
//
// Generated By:JAX-WS RI IBM 2.1.1 in JDK 6 (JAXB RI IBM JAXB 2.1.3 in JDK 1.6)
//
package com.myaccount.account;
...
@WebService(name = "AccountService", targetNamespace = "http://www.myaccount.com/account")
...
public interface AccountService {
    /**
     *
     * @param days
     * @param account
     * @return
     *     returns float
     */
    @WebMethod(action = "computeAccountAverage")
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "computeAccountAverage", targetNamespace = "http://www.myaccount.com/account",
         className = "com.myaccount.account.ComputeAccountAverage")
    @ResponseWrapper(localName = "computeAccountAverageResponse", targetNamespace = "http://www.myaccount.com/account",
         className = "com.myaccount.account.ComputeAccountAverageResponse")
    public float computeAccountAverage(
        @WebParam(name = "account", targetNamespace = "")
        Account account,
        @WebParam(name = "days", targetNamespace = "")
        int days);

}
```

This code example is a Java interface, not merely a Java class. The @WebService annotation is present in this Java interface. It is important to know that this example is not the same as the generated @WebServiceClient class, com.myaccount.account.AccountWSDLService. This class is not an interface and is actually not needed in your SCA application.

4. Now that you have Java interface either by generating the Java interface from a WSDL file or you have an existing Java interface, you are ready to develop your SCA client from the Java interface.
5. Place the @Reference annotation on a public or protected field or setter, with the same type as your Java interface.

```
package com.myaccount.client;

import bank.process.BankProcess;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.Service;

import com.myaccount.account.*;

@Service(BankProcess.class)
public class AccountClientComponent implements BankProcess {

   // Note the type, 'AccountService', is the Java interface generated from
   // from the WSDL portType
    private AccountService accountServiceRef;

    //
    // Injected by the SCA container
    //
```

```
        @Reference
        public void setAccountServiceRef(AccountService accountServiceRef) {
            this.accountServiceRef = accountServiceRef;
        }

        public String someMethod(String input) {

            //... some business logic ...

            //  We'll show a simple example of JAXB API usage
            ObjectFactory factory = new ObjectFactory();
            Account account = factory.createAccount();
            account.setAccountNumber(4);
            account.setAccountID("CHECKING");

            int days = 5;

            float avg = accountServiceRef.computeAccountAverage(account, days);

            //... the rest of the business logic ...
        }
    }
```

6. Create a component using the component implementation. When using a WSDL portType interface, you must create component definitions in the composite definition that references the original portType along with the SCA Java implementation. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. Shown here are the aspects of component creation that are common for all component definitions using the implementation developed in this example. This example also includes bindings that you can modify or omit for other components using this component implementation.

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
     targetNamespace="http://bank.process/customer"
         name="bpComposite">

      <component name="BankProcessComponent">

          <implementation.java class="com.myaccount.client.AccountClientComponent"/>
         <!-- The @name attribute corresponds to the setter that is annotated with the @Reference annotation. -->

         <reference name="accountServiceRef">
              <!-- This statement specifies the QName of the WSDL portType,
                    "http://www.myaccount.com/account#AccountService" in the syntax as
                   illustrated in the interface.wsdl statement.  -->

             <interface.wsdl interface="http://www.myaccount.com/account#wsdl.interface(AccountService)" />
             <binding.ws uri="http://localhost:9080/BankingComponent/AccountService"/>

  <!-- This example uses the SCA web services binding.  However, it does not matter which specific binding
 you choose.  You can also choose to use the SCA default binding or the SCA EJB binding.  -->

          </reference>
     </component>
</composite>
```

7. Configure the composite definition when starting with a Java interface.

   The following snippet is another example of the syntax if you develop an SCA client starting with a Java interface rather than with a WSDL portType. To simplify this example, use the same AccountService Java interface from the previous step but in this case, assume that it was not generated from a WSDL file.

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
     targetNamespace="http://bank.process/customer"
         name="bpComposite">

      <component name="BankProcessComponent">

         <implementation.java class="com.myaccount.client.AccountClientComponent"/>
```

```
        <!-- The @name value corresponds to the setter that is annotated with the @Reference annotation. -->
               <reference name="accountServiceRef">

             <!-- Because the runtime can introspect the interface, it is unnecessary to specify
                  the interface.java in the composite definition. This is what the interface looks like if you include it.

                        <interface.java interface="com.myaccount.account.AccountService"/>  -->

              <!-- The SCA binding type is omitted. It does not matter which specific SCA binding you choose. -->
        </reference>
    </component>
</composite>
```

8. After a component is defined as part of a deployable composite, either directly or recursively through use of one or more layers of components with composite implementation, you are now ready to deploy the SCA component by creating a SCA business level application.

Developing asynchronous SCA services and clients

You can create applications that use Service Component Architecture (SCA) OASIS specifications to asynchronously run request-response services.

Before you begin

Note: SCA OASIS specifications support the asynchronous running of request-response services. This enables a client thread to continue doing other work while the service runs. You can use SCA OASIS annotations and APIs in Java interfaces to enable asynchronization in services.

To learn about asynchronous invocations of SCA services, see the SCA OASIS Java Common Annotations and APIs specification. For a list of common annotations in SCA OASIS specifications, see `http://docs.oasis-open.org/opencsa/sca-j/javadoc/index.html`.

About this task

An SCA client and an SCA service have independent capabilities for asynchronous invocation which can be intermixed. This means that a client can synchronously run an asynchronous service or asynchronously run a synchronous service. Typically, a client asynchronously runs an asynchronous service.

When developing an asynchronous service and client, consider the following:

- Use the SCA OASIS annotation `@AsyncInvocation` in the Java service interface to enable asynchronization.
- Use the SCA OASIS annotation `@AsyncFault` in the Java service interface for exception errors.
- In the composite definition, use the `implementation.java` type for the SCA component.
- Asynchronous interfaces are supported by `binding.sca` only.

Topics in the procedure cover developing SCA services and clients that run asynchronously.

Procedure

1. Develop an asynchronous SCA service.
2. Develop an asynchronous SCA client.

What to do next

Deploy the SCA composites in a business-level application.

For SCA OASIS applications, an `sca-contribution.xml` file is required for deployable composites in the `META-INF/` directory, and not in a subdirectory.

Developing asynchronous SCA services

You can create services that use Service Component Architecture (SCA) OASIS specifications to asynchronously run request-response services.

Before you begin

To learn about asynchronous invocations of SCA services, see the SCA OASIS Java Common Annotations and APIs specification. For a list of common annotations in SCA OASIS specifications, see `http://docs.oasis-open.org/opencsa/sca-j/javadoc/index.html`.

About this task

To develop an asynchronous SCA service, you can create three files:

- A Java service interface that uses the SCA OASIS `@AsyncInvocation` annotation and, for exception errors, the `@AsyncFault` annotation
- A Java service implementation that has an `@Service` annotation which refers to the service interface
- An SCA OASIS composite

Procedure

1. Create an asynchronous service interface.

   You can derive an interface from the business interface between the client and the service. For example, suppose that the service has the following business interface:

   ```
   package broker;
   public interface StockQuote {
   float getPrice(String symbol) throws UnknownSymbolException;
   }
   ```

   Copy the business interface and modify it to create an equivalent asynchronous service interface:

   ```
   package broker.impl;
   import broker.UnknownSymbolException;
   import org.oasisopen.sca.ResponseDispatch;
   import org.oasisopen.sca.annotation.AsyncFault;
   import org.oasisopen.sca.annotation.AsyncInvocation;

   @AsyncInvocation
   public interface StockQuote {
       @AsyncFault(UnknownSymbolException.class)
       void getPriceAsync(String symbol, ResponseDispatch<Float> dispatch); }
   ```

   An asynchronous service interface is a contract between the SCA container and the service implementation. It is not used by the client.

   Use the `@AsyncInvocation` annotation on the interface to indicate it is asynchronous. Derive each method from its equivalent method in the business interface as follows:

   - Append the characters `Async` to the method name.
   - Change the return type to `void`.
   - Add a `ResponseDispatch` argument which is typed by the method's original return type.
   - Move exceptions from the `throws` clause to `@AsyncFault` annotations.

   For simplicity, use the same simple name for the asynchronous service interface as for the business interface, but specify a different package name. By default, an SCA service name is the simple name of its interface, so using the same interface names helps to ensure that a consistent SCA service name is used.

2. Create the asynchronous service implementation.

   For example, create an implementation that references the `StockQuote` SCA service interface:

   ```
   package broker.impl;
   import broker.UnknownSymbolException;
   import org.oasisopen.sca.ResponseDispatch;
   import org.oasisopen.sca.annotation.Service;
   @Service(StockQuote.class)
   ```

```
public class StockQuoteImpl {
    public void getPriceAsync(String symbol, ResponseDispatch<Float> dispatch) {
         if (!isValidSymbol(symbol))
             dispatch.sendFault(new UnknownSymbolException());
         else
             dispatch.sendResponse(getPrice(symbol));
     }
     private boolean isValidSymbol(String symbol) {
         // fill in
     }
     private Float getPrice(String symbol) {
         // fill in
     }
}
```

The `@Service` annotation must refer to the asynchronous service interface.

Each method is passed a `ResponseDispatch` object which must be used to return a response to the client. The example shows the response being sent inside the method body but this is not required. The method can save the input arguments, including the `ResponseDispatch` object, for another thread to handle.

Do not perform a long-running computation directly in the method body because this might cause the client to receive a timeout exception. Instead, queue the work to another thread. One approach for queueing work to another thread is to use an asynchronous bean.

3. Create a composite file.

   For example, create an SCA OASIS composite that defines an `implementation.java` component which uses the implementation `StockQuoteImpl` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
           targetNamespace="http://www.example.com" name="StockQuoteComposite">
    <component name="StockQuoteComponent">
        <implementation.java class="broker.impl.StockQuoteImpl"/>
    </component>
</composite>
```

   Asynchronous services must use the `binding.sca` binding type because it is the only binding that supports asynchronous invocation. However, `binding.sca` is the default binding so you do not need to include it in the composite file.

What to do next

Develop an asynchronous SCA client to run an SCA service.

Developing asynchronous SCA clients

You can create clients that use Service Component Architecture (SCA) OASIS specifications to run asynchronously.

Before you begin

To learn about asynchronous invocations of SCA services, see the SCA OASIS Java Common Annotations and APIs specification. For a list of common annotations in SCA OASIS specifications, see `http://docs.oasis-open.org/opencsa/sca-j/javadoc/index.html`.

About this task

An SCA client can synchronously run an asynchronous service or asynchronously run a synchronous service. Typically, a client asynchronously runs an asynchronous service.

To develop an asynchronous SCA client, you can create three files:

- A Java client interface
- A Java client implementation that has an `@Reference` annotation
- An SCA OASIS composite

To dispatch client-side asynchronous requests, you can use a default SCA work manager.

A few limitations apply when using asynchronous interfaces from an SCA client:

- If the client application or its host server is stopped, responses to active asynchronous requests are lost, and are not delivered when the application is restarted.
- Because the SCA container invokes an asynchronous request from a separate thread, the client must not modify any arguments passed to the request until after it receives the response.
- Asynchronous invocation is not supported with services obtained using the `org.oasisopen.sca.client.SCAClientFactory` interface.
- A client can synchronously invoke an asynchronous service. The SCA container waits for up to 120 seconds for a response from the asynchronous service.

Procedure

1. Create an asynchronous client interface.

   You can derive an interface from the business interface between the client and the service. For example, suppose that the service has the following business interface:

   ```
   package broker;
   public interface StockQuote {
   float getPrice(String symbol) throws UnknownSymbolException;
   }
   ```

   Copy the business interface and modify it to create an equivalent asynchronous client interface:

   ```
   package broker.client;
   import broker.UnknownSymbolException;
   import java.util.concurrent.Future;
   import javax.xml.ws.AsyncHandler;
   import javax.xml.ws.Response;
   public interface StockQuote {
       float getPrice(String symbol) throws UnknownSymbolException;
       Response<Float> getPriceAsync(String symbol);
       Future<?>getPriceAsync(String symbol, AsyncHandler<Float> callback);
   }
   ```

   An asynchronous client interface is a contract between the client and the SCA container. It is not used by the service.

   An asynchronous client interface has additional methods to support polling or callback delivery. Derive the polling method from its equivalent method in the business interface as follows:

   - Append the characters `Async` to the method name.
   - Change the return type to `javax.xml.ws.Response`.

   Derive the callback method from its equivalent method in the business interface as follows:

   - Append the characters `Async` to the method name.
   - Change the return type to `java.util.concurrent.Future`.
   - Add a `javax.xml.ws.AsyncHandler` argument which is typed by the return type of the original method.

2. Create the asynchronous client implementation.

   For example, create an implementation that uses the callback interface:

   ```
   package broker.client;
   import broker.UnknownSymbolException;
   import java.util.concurrent.ExecutionException;
   import javax.xml.ws.AsyncHandler;
   import javax.xml.ws.Response;
   import org.oasisopen.sca.annotation.Reference;
   public class StockQuoteClientImpl {
       @Reference
       public StockQuote quoteService;

       public void getStockPrice(String symbol)  {
           CallbackHandler callback = new CallbackHandler(symbol);
           quoteService.getPriceAsync(symbol, callback);
   ```

```
    }

    private class CallbackHandler implements AsyncHandler<Float>{
        private String symbol;
        public CallbackHandler(String symbol) {
            this.symbol = symbol;
        }
       public void handleResponse(Response<Float>arg0) {
           try {
               Float price = arg0.get();
               // Process the response
           } catch (ExecutionException e) {
              if (e.getCause() instanceof UnknownSymbolException) {
                  // Process the exception
              }
           } catch (Throwable t) {
               // Process the exception
           }
    }
}
```

3. Create a composite file.

   For example, create an SCA OASIS composite that defines an `implementation.java` component which uses the implementation `StockQuoteClientImpl` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
           targetNamespace="http://www.example.com" name="StockQuoteClientComposite">
    <component name="StockQuoteClientComponent">
        <implementation.java class="broker.client.StockQuoteClientImpl"/>
    </component>
</composite>
```

   Asynchronous clients must use the `binding.sca` binding type because it is the only binding that supports asynchronous invocation. However, `binding.sca` is the default binding so you do not need to include it in the composite file.

4. Configure the default SCA work manager.

   The SCA container uses the default SCA work manager to dispatch client-side asynchronous requests. See Configuring the default SCA Work manager for the SCA layer.

What to do next

Deploy the SCA client and service in a business-level application.

Using business exceptions with SCA interfaces

You can implement exceptions for remotable interfaces in the Service Component Architecture (SCA) environment to provide additional flow of control for error conditions to meet the needs of your business application.

About this task

To develop SCA service implementations, you can use either a top-down development approach starting with an existing Web Services Description Language (WSDL) file or you can use a bottom-up development approach starting from an existing Java interface or implementation. When using either the top-down or bottom-up development methodologies, you can use tools to map business exceptions on remotable interfaces.

In order to achieve the SOA goal of providing an interoperable platform that is both language and technology neutral, the SCA runtime environment takes an XML-centric view of interfaces and data. When working with Java code, the Java API for XML-Based Web Services (JAX-WS) standard is used to define the mapping between Java code and the XML-based Web Services Description Language (WSDL) file. This mapping also includes the Java programming model with respect to exceptions. Exceptions for

remotable interfaces in the SCA environment is defined by the JAX-WS specification. This topic describes the best practices for using business exceptions with SCA interfaces.

Differences between business exceptions and fault beans

To better understand the implications of implementing business exceptions in an SCA environment, it is helpful to understand differences between an exception and a fault bean.

The JAX-WS specification distinguishes between a checked exception and the fault bean that it wrappers. However, this distinction might not be clear because a single class can serve the checked exception and the fault bean functions, especially when you use the bottom-up approach of developing an SCA service starting with a Java interface. When you use the top-down development approach of developing an SCA service starting with a WSDL file, section 2.5 of the JAX-WS specification describes the wrapper pattern for how the fault message maps to a Java checked exception that wraps a fault bean. The fault bean maps to the fault element and in SCA environments, the mapping is defined by Java Architecture for XML Binding (JAXB) data binding. The fault bean represents the cross-platform view of the fault message data and includes a schema description. You can use the Java exception within the Java runtime environment and as part of the Java programming model. However, the exception is not part of the interoperable data representation.

When developing SCA services using the bottom-up approach, the distinction between an exception, the fault bean, and the mapping from Java to WSDL or XSD schema is clear if you follow the wrapper pattern described in section 2.5 of the JAX-WS specification. If you have existing Java exceptions, use the standard mapping defined in section 3.7 of the JAX-WS specification for service specific exceptions. In SCA environments, these service specific exceptions are referred to as business exceptions. The mapping for the business exceptions is different than the mapping described in section 2.5 of the JAX-WS specification. Because this wrapper pattern only applies for certain exceptions, this approach has limitations when using the bottom-up development approach. The possible limitations of using the wrapper pattern to implement error handling when using bottom-up development of SCA applications provides additional reasons to consider the advantages of the best practice of top-down development of SCA applications.

Top-down development, starting from a WSDL operation with fault messages

It is a best practice to use the top-down methodology to develop SCA service implementations because this approach leverages the capabilities of the XML interface description and provides a greater ease in interoperability across platforms, bindings, and programming languages. A WSDL operation can be defined, along with one or more fault messages, provided each fault message is defined in terms of a fault element. When the **wsimport** command-line tool is used to generate Java code the tool generates Java exception code that wraps a fault element in the format specified by the Java API for XML-Based Web Services (JAX-WS) specification, section 2.5.

Bottom-up development, starting from a Java operation with throws clause

Bottom-up development of SCA services occurs when you start with existing Java code. Using this development approach, do not design a remotable interface that might cause a technology exception such as java.sql.SQLException. This exception is more appropriate for a local interface rather than a coarse-grained remotable interface.

Procedure

1. For top-down development of SCA applications, implement a wrapper pattern for business exceptions. The wrapper pattern is based on section 2.5 of the JAX-WS specification.
   a. Obtain your WSDL file; for example:

```
<wsdl:types>
    ...
    <element name="errorCode" type="xsd:int"/>
    ...
</wsdl:types>
```

```
<wsdl:message name="BadInputMsg">
    <wsdl:part element="tns:errorCode" name="parameters"/>
</wsdl:message>

<wsdl:portType name="GuessAndGreet">
    <wsdl:operation name="sendGuessAndName">
            <wsdl:input.../>
            <wsdl:fault message="tns:BadInputMsg" name="BadInputMsg"/>
```

b. Generate the Java artifacts using the **wsimport** tool. You can define the fault according to section 2.5 of the JAX-WS specification; for example:

```
Interface
   public Person sendGuessAndName(...) throws BadInputMsg;
```

c. Wrap an exception in a fault; for example:

```
import javax.xml.ws.WebFault;

@WebFault(name = "errorCode", targetNamespace = "....")
public class BadInputMsg extends Exception
{
    private int faultInfo;

    public BadInputMsg(String message, int faultInfo) {
        super(message);
        this.faultInfo = faultInfo;
    }

    public BadInputMsg(String message, int faultInfo, Throwable cause) {
        super(message, cause);
        this.faultInfo = faultInfo;
    }

    public int getFaultInfo() {
        return faultInfo;
    }
}
```

2. For bottom-up development of SCA applications, implement or convert the exception to follow the wrapper pattern or use the default mapping for of a JAX-WS service specific exception.

   If you have a Java business exception, the complexity of this scenario increases, especially if your exception wraps fault data. For example, the exception wraps data such as an error code or an object that it needs to provide to the client that receives the exception. In this scenario, there are two options:

   - Convert the Java business exception to follow the wrapper pattern as described in section 2.5 of the JAX-WS specification.

     Using the wrapper pattern for the exception enables the exception to map easily from the WSDL to Java code format and then from the Java code to WSDL format. If you modify the exception to follow the wrapper pattern, you can use the **wsgen** tool to convert from Java code to WSDL and later use the **wsimport** tool to convert from WSDL to Java code, the exception is similar to the one that you modified. To achieve this end goal, you must perform the following steps:

     a. Add constructors that take the fault bean as input parameters.
     b. Implement a public getFaultInfo() method that returns the fault bean.
     c. Add the @javax.xml.ws.WebFault annotation. See the example that wraps an exception in a fault.

   - Use the default mapping of a JAX-WS service specific exception or business exception as described in section 3.7 of the JAX-WS specification.

     If you use the **wsgen** command-line tool to generate the WSDL, the tool uses this pattern for business exceptions. If you do not generate the WSDL file before deployment, the application server runtime environment implicitly generates the business exception using this pattern.

     Use this option when you:

     – cannot change the exception class to follow the JAX-WS wrapper pattern.
     – rely on the runtime environment to map the Java code into WSDL such as declaring a `<binding.ws>` binding on a service that is deployed without a WSDL file.

Either of these options work without any additional complexity as long as the exception does not contain fault data.

For exceptions with fault data, the data is handled correctly for each field that contains a public getter or setter method. However, data is lost without a getter or setter pair. In other words, serialize or deserialize the exception by viewing it as a Java code.

When using this second option, the following items are important:

- The supported fault pattern is not easily determined. One exception with fault data and also with the getter and setter methods is that some are handled correctly while others are not. Running the **wsgen** tool at development time generates the schema based on the exception getter methods without assuring that the corresponding setter methods exist in order to populate the exception during unmarshalling.
- If you run **wsimport** tool against the generated WSDL, you get a different exception class. Your client and service programming model are different which might confuse the Java programmer. However, this generated Java exception follows the pattern described in the JAX-WS specification in section 2.5. You might need to add customization for JAXB data binding in order to generate the client. The results can produce exception names similar to `MyException_Exception`.
- Although the service-specific exception pattern is described in section 3.7 of the JAX-WS specification, not all details for the pattern are specified. Other software implementing JAX-WS might implement the pattern differently. This is not critical, since the WSDL file is interoperable across platforms.

Example 1: No fault

The following examples illustrates using the bottom-up development of SCA applications and using the business exception mapping as described in section 3.7 of the JAX-WS specification.

The string message is the fault in this example, and it is serialized and deserialized successfully.

```
public class RealSimpleException extends Exception {
    public RealSimpleException(String message) {
        super(message);
    }
    public RealSimpleException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

This example works correctly because the string `userdata` fault has associated public getter and setter methods. The string message is also handled correctly.

```
public class TestException extends Exception {

    private String userdata;

    public TestException(String message) {
        super(message);
    }

    public TestException(String message, String userdata) {
        super(message);
        this.userdata = userdata;
    }

    public String getUserdata() {
        return userdata;
    }

    public void setUserdata(String userdata) {
        this.userdata = userdata;
    }

}
```

This example does not work correctly because the errorCode fault data does not have a setter method. The SCA runtime is not able to correctly determine how to populate the exception with this fault data. The exception occurs, but it is displayed with data loss.

```
package java.sql;

public class SQLException extends Exception ... {
 ...
 public SQLException(String theReason, String theSQLState, int theErrorCode) ...

 public int getErrorCode()
}
```

What to do next

Increase portability of your exception classes in top-down development

One issue that you might encounter with Java exceptions generated from WSDL in the top-down manner is that the fault bean might not be Java-serializable. In other words, the fault bean might not implement java.io.Serializable. This does not present a problem if the bindings that your application uses are in XML wireformat because, in that scenario, XML serialization is used instead of Java serialization. However, XML serialization limits the usefulness of your exception class. Therefore, it might not be suitable to use with SCA binding configurations using a wireformat with Java serialization, or in other contexts in which Java serialization is used to serialize and deserialize an exception.

To avoid this limitation, annotate your schema definition for the fault element type with a JAXB customization designed for this purpose. When this customization is present, the JAXB type that is generated, that corresponds to the fault bean, is marked as implementing the java.io.Serializable interface, and is therefore Java serializable, in addition to being serializable to XML, because the class is also still a conforming JAXB type.

Example:

```
<schema targetNamespace="http://com.mycompany/banking/" jaxb:version="2.0"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns="http://www.w3.org/2001/XMLSchema">
   <annotation>
     <appinfo>
       <jaxb:globalBindings>
         <jaxb:serializable uid="1"/>
      </jaxb:globalBindings>
     </appinfo>
   </annotation>

    <!-- Continue with the rest of the schema definition-->

 </schema>
```

SCA programming tip for binding neutrality

The JAX-WS defined mapping between exceptions in Java and an XML wireformat relies on the use of the fault bean to pass back data from the service provider throwing the exception to the client that is catching it. Normal Java-centric mechanisms, such as exception chaining, are not preserved in mapping between the Java application and the wire. Therefore, the best way to write application code that can be used across binding configurations using either of a Java serialization wireformat or an XML wireformat is to rely exclusively on the fault bean for communicating useful application level data relating to the exception.

Even though using a chained exception across a binding that uses a Java-serialization based wireformat works fine, using a chained exception across a binding using XML wireformat might not work. Therefore relying on a chained exception would not be a recommended practice in an SCA environment, because SCA strives to provide a binding-neutral programming model.

Considerations for developing SCA applications using EJB bindings

When developing Service Component Architecture (SCA) applications that you intend to use with Enterprise JavaBeans (EJB) bindings, keep in mind that the SCA EJB binding is architected in a Java-centric manner, in contrast to the XML-centric implementations of the SCA default binding and the SCA web services binding.

The EJB transports marshal and unmarshal application data into the wire format by using Java serialization, whereas the web services and default bindings use XML serialization. This difference also affects the programming model in that the SCA clients and implementations using the EJB binding must use java.io.Serializable types, in contrast to the Java Architecture for XML Binding (JAXB) data types-based programming model that is used for the SCA default and web services bindings.

SCA reference

In this case, you have an existing EJB that you want to invoke with an SCA client using a reference that is configured with an EJB binding.

When you develop an SCA client that will invoke an existing EJB using the SCA EJB binding, you must use a Java interface when developing the SCA client rather than using a WSDL interface. The EJB binding marshalling of application data into the wire data format is performed using Java serialization, not XML serialization as defined by JAXB.

To learn more about SCA references, read about developing SCA service clients. However, when you are using the SCA EJB binding, the information in this topic takes precedence.

Because you obtain the Java interface and parameter types from the EJB provider for use in your client, you do not have to worry about the effects of marshalling and unmarshalling when writing your client. However, when you provide these data types across new services, problems can occur if you pass these data types across new services, because they might not serialize correctly over other bindings, such as the default binding, because of the difference in Java serialization and the JAXB XML marshalling and unmarshalling.

The following example illustrates the problematic scenario of starting with an existing EJB interface and using a Java serializable data type that does not serialize well using JAXB marshalling and unmarshalling.

```
public interface NameService extends javax.ejb.EJBObject {

    public String computeName(Person p) throws RemoteException;
}
// This snippet is intended as an example of a type that is problematic.
public class Person implements java.io.Serializable {

    private int code;
    private String name;

    // The code field must be passed into constructor.  However, this causes problems for
    // for SCA default and web services bindings that use JAXB marshalling/unmarshalling.

    public Person(int code) {
        this.code = code;
    }

    public Person() {
    }

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

}
```

The following SCA client A example works correctly. The `Person` object that is instantiated directly in the `ClientAImpl` implementation is correctly marshalled to invoke the EJB with a remote interface of `NameService`.

```
// Client A  ClientAImpl.java
 .....
@Reference
public NameService nameService

public someClientMethod() {
    // No problem when Person object is instantiated by client
    Person person = new Person(5);
    String name = nameService.computeName(person);
}
```

In contrast, the following example demonstrates the problem with the `Person` type. The client code has been refactored so that it contains a reference to `NameService`, and it obtains the `Person` object that is passed into the `computeName` method over a new remotable interface, rather than constructing it directly.

```
// Problem client interface

import org.osoa.sca.annotations.Remotable;
@Remotable
public interface PersonFilter {
     boolean filterPerson(Person p);
}


// Problem client implementation

@Service(PersonFilter.class)
public class PersonFilterImpl implements PersonFilter {
     @Reference
     public NameService nameService

     boolean filterPerson(Person p) {
     // ...  business logic
    String name = nameService.computeName(person);
     // ...  business logic
     }
}
```

If the `PersonFilterImpl` class receives a `Person` object from the client over the `PersonFilter` interface and the implementation is invoked using the SCA default binding, the data is not handled correctly. The default binding does not preserve the `code` field of the `Person` object that is passed to the `PersonFilterImpl` class.

For a class without JAXB annotations, JAXB marshalling and unmarshalling preserves JavaBeans properties, but not private data such as the `code` field, which does not have a setter and is only established in the constructor. When the `Person` object is passed to the `NameService` EJB, the `code` value is set to the default value of `0` regardless of what the `PersonFilter` client passed to the `PersonFilterImpl` class.

If the `Person` type was written in the JavaBeans style with getters and setters for all important data, then this type works correctly in the example for the `PersonFilterImpl` client. However, if you are consuming an existing EJB, you do not have control over the types it already uses on its interface. Not all existing Java types are optimal for SCA Java programming. To address the problems in this example, you must create a new type for use on the `PersonFilter` interface and translate the data for this type into a `Person` object within the `PersonFilterImpl` class which directly invokes the EJB with the remote interface `NameService`.

In this example, if the `PersonFilter` interface was defined as a local interface, then the concerns with preserving data integrity do not apply. The runtime environment performs pass-by-reference semantics across local interfaces that are appropriate for tightly-coupled clients and services such that no data is copied.

SCA service

If you write a new SCA service and intend to expose it over the SCA EJB binding so that an EJB client can invoke the service, it is a best practice to develop the SCA service using the top-down methodology starting with an existing WSDL file or XSD schema and generating the JAXB classes that are used to write the service implementation. Using this approach, you can easily address the differences between Java serialization and JAXB marshalling and unmarshalling by specifying that the generated JAXB classes are Java serializable.

To enable the generated JAXB classes to work correctly over the SCA EJB binding, add the `serializable` customization to the schema definition so that the generated JAXB classes are Java serializable and implement the `java.io.Serializable` interface. For example:

```
<schema targetNamespace="http://com.mycompany/banking/" jaxb:version="2.0"
 xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
 xmlns="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <appinfo>
      <jaxb:globalBindings>
        <jaxb:serializable uid="....."/>
     </jaxb:globalBindings>
    </appinfo>
  </annotation>

   <!-- Continue with the rest of the schema definition-->

</schema>
```

As a result, you can use your Java implementation with the generated JAXB data types over the EJB binding, which uses Java serialization. At this point, because you have the generated JAXB artifacts, you can also use your Java implementation with the generated JAXB data types over the SCA default and SCA web services bindings, which use XML serialization as defined by JAXB.

If you develop your SCA service using the bottom-up approach starting with Java code, you must use types that implement the `java.io.Serializable` interface as required when writing an EJB. See the developing SCA services with existing Java code documentation for more information regarding requirements for your user-defined types. Also, see the SCA reference section to learn how to avoid problems with your user-defined types when using EJB bindings because of the differences between Java serialization and JAXB marshalling and unmarshalling.

Specifying bindings in an SCA environment

After you develop an Service Component Architecture (SCA) component, you can use bindings to specify how SCA services and references enable the component to communicate with other applications.

About this task

Services and references enable a component to communicate with other applications. By design, however, the SCA services and references say nothing about how this communication occurs. Bindings are used to determine how a component communicates with the world outside its domain. SCA services use bindings to describe the access mechanism that clients must use to call the service. SCA references use bindings to describe the access mechanism that is used to call a service. Depending on what the SCA component is communicating with, a component might or might not have explicitly specified bindings.

The product supports the following binding types:

- SCA binding

The SCA binding is also referred to as the default binding. It is the binding that is used when no other binding is specified for configuration of a component reference or service. It is the natural binding to use when your SCA client invokes an SCA service in the same domain.

SCA default bindings are not compatible across implementations by architecture; however, some interoperability scenarios are enabled between Open SCA and Classic SCA as implemented in the BPM Suite of products like WebSphere Process Server.

Components communicating within the same domain only need to explicitly configure a default binding on a service or reference when there is at least one non-default binding, such as the SCA web services binding or the SCA EJB binding, that is also configured.

- Web service binding

  The SCA web services binding applies to the services and references of components. The web service binding is designed for SOAP-based Web Services-Interoperability (WS-I) compliant web services. This binding defines the manner in which a service is made available as a web service, and in which a reference can invoke or access a web service. The web service binding enables SCA applications to expose SCA services as web services to external clients that might or might not be implemented as an SCA component. This binding is a Web Services Description Language (WSDL)-based binding which means that the web service binding either references an existing WSDL binding or enables you to specify enough information to generate a WSDL file. When an existing WSDL binding is not referenced, you can generate a WSDL binding. You can further customize an SCA web services binding using SCA policy sets.

  Web services technology plays an important role in most SOA solutions relevant today, including SCA. The SCA web service binding type enables SCA applications to expose services as web services to external clients as well as enabling SCA components access to external web services. External clients that access SCA services exposed as web services may or may not be implemented as an SCA component. You can use the web service binding element `<binding.ws>` within either a component service or a component reference definition. When the web service binding is used with a component service, this binding type enables clients to access a service offered by a particular component as a web service. When the web service binding is used with a component reference, components in an SCA component can consume an external web service and access as if it was any other SCA component. Only WSDL Version 1.1 is supported.

- EJB binding

  EJB session beans are a common technology used to implement business services. The ability to integrate SCA with services based on session beans is useful because it preserves the investment incurred during the creation of those business services, while enabling the enterprise to embrace the newer SCA technology in incremental steps. The simplest form of integration is to simply enable SCA components to invoke session beans as SCA services. There is also a need to expose services such that they are consumable by programmers skilled in the EJB programming model. This enables existing session bean assets to be enhanced to exploit newly deployed SCA services without the EJB programmers having to learn a new programming model.

  The SCA EJB binding enables SCA to integrate with existing Java EE applications. It exposes SCA services as stateless session beans to external clients. The binding element `<binding.ejb>` is used within a component service or component reference definition. Support is provided for the EJB binding when using both 2.x and 3.0 EJB styles for both the SCA service and reference.

- JMS binding

  Use the SCA Java Message Service (JMS) binding to compose and assemble SCA services from new and existing services that are available using the JMS programming interface. By taking advantage of the ability to integrate SCA with existing services based on JMS, you can preserve your investment in enterprise messaging technology, while enabling the enterprise to embrace the new SCA technology in incremental steps.

  The SCA JMS binding enables an open, implementation neutral, service-oriented description of the SCA service assembly and composition. Use the binding element `<binding.jms>` within a component service or component reference definition. SCA services using the JMS binding are exposed using the Java EE Connector Architecture (JCA)-based messaging provider in WebSphere Application Server.

The product supports the default messaging provider or WebSphere MQ as the messaging engine.

- Atom binding

  Use the Atom binding to work with services that provide or consume entries described in the Atom Syndication Format and Atom Publishing Protocol. An SCA component can reference existing external web feeds defined using the Atom protocol and work with them inside a Java implementation. Also, you can use the Atom binding to compose new services and expose them as an Atom feed.

  The product supports the Atom binding for OSOA, but not for OASIS.

- HTTP binding

  Use the HTTP binding to expose SCA services for consumption by remote JavaScript-enabled web browser clients. Using this binding, clients can invoke Remote Procedure Calls (RPC) to server-side SCA components.

  The product supports the HTTP binding for OSOA, but not for OASIS.

Procedure

1. Select a binding type to use for an SCA component.
   - Use the SCA default binding when you want to invoke an SCA service from an SCA client.
   - Use the SCA web services binding to specify that an SCA service is made available as a web service or an SCA reference can invoke a web service.
   - Use the SCA EJB binding to integrate SCA with services based on session beans.
   - Use the SCA JMS binding to compose and assemble SCA services from new and existing services.
   - Use the Atom binding to expose collections of data as an Atom feed or to reference existing external Atom feeds.
   - Use the HTTP binding with a wire format of JSON-RPC to expose services to remote web browser clients.
2. Configure the selected binding and use it in an SCA component or application.

Results

SCA components can use the configured bindings to communicate with other SCA services and references.

What to do next

Deploy the SCA component or application.

Configuring the SCA default binding

You can configure the Service Component Architecture (SCA) default binding for services and references.

About this task

Bindings determine how a component communicates with the world outside its domain. Services use bindings to describe the access mechanism that clients must use to call the service. References use bindings to describe the access mechanism used to call a service. The SCA binding is also referred to as the default binding. The default binding is the binding that is used when no other binding is specified for a configuration of a component reference or service. Use this binding when an SCA client invokes an SCA service in the same domain. It is not intended to be interoperable in any way with other implementations of SCA runtime environments.

Procedure

- Configure an SCA service with the SCA default binding.

If the service is only exposed over the default binding, then you do not need to explicitly add the <binding.sca> element because this binding is default binding for SCA. If your SCA service has more than one binding and the SCA default binding must be one of them, you must specify the <binding.sca> element in the composite definition.

- Configure an SCA reference with the SCA default binding.

  For the reference, you also do not need to specify the <binding.sca> element. For an reference with a default binding, the reference specifies a target attribute indicating the target service. To indicate the target at the reference, specify target=componentName/serviceName. If only one service exists for the service component, then you only need to specify the componentName; for example: target=ComponentName.

Results

You have implicitly or explicitly configured the SCA default binding for your SCA service or reference.

Example

The following examples illustrate multiple scenarios for configuring the SCA default bindings.

Top level composite with SCA service binding

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://mysca/samples" name="MyComposite">

      <component name="HelloWorldServiceComponent">
         <implementation.java class="test.HelloWorldImpl"/>
      </component>

   </composite>
```

Top level composite with SCA reference binding

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://mysca/samples"
       name="ClientComposite">


      <component name="ClientComponent">
        <implementation.java class="test.GreetingsServiceImpl"/>
        <reference name="helloWorldService" target="TargetComponent"/>
      </component>

   </composite>
```

OR:

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://mysca/samples"
        name="ClientComposite">


      <component name="ClientComponent">
        <implementation.java class="test.GreetingsServiceImpl"/>
        <reference name="helloWorldService" target="TargetComponent/HelloWorld"/>
        <!-- compName/serviceName -->
        </component>

   </composite>
```

Top level composite with SCA service binding with transaction policy attribute defined

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://neworder/sca/jdbc"
       name="NewOrderComposite">
```

```
      <component name="NewOrderServiceComponent">
         <service name="NewOrderService" requires="propagatesTransaction.false"/>
         <implementation.java class="neworder.sca.jdbc.NewOrderServiceImpl" requires="managedTransaction.local"/>
      </component>

   </composite>
```

Top level composite with SCA service binding supporting WSDL interface

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://mysca/samples"
         name="ClientComposite">

        <component name="ClientComponent">
           <service name="HelloWorldService">
             <interface.wsdl interface="http://helloworld#wsdl.interface(HelloWorld)"/>
           </service>
          <implementation.java class="test.HelloWorldImpl"/>
         </component>

   </composite>
```

Top level composite with SCA reference binding supporting WSDL interface

```
<?xml version="1.0" encoding="UTF-8"?>

    <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://mysca/samples" name="ClientComposite">


       <component name="ClientComponent">
         <implementation.java class="test.GreetingsServiceImpl"/>
         <reference name="helloWorldService" target="MyServiceComponent">
            <interface.wsdl interface="http://helloworld#wsdl.interface(HelloWorld)"/>
         </reference>
       </component>

</composite>
```

Intra composite over SCA default binding

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://mysca/samples"
         name="Service1Composite">

       <component name="HWServiceComponent">
         <implementation.java class="test.HelloWorldImpl"/>
         <reference name="component2Ref" target="Component2"/>
      </component>


        <component name="Component2">
        <implementation.java class="test.Component2Impl"/>
        </component>

   </composite>
```

SCA Service with workManager specified for the service

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  xmlns:wm="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
         targetNamespace="http://mysca/samples" name="Composite2">


      <component name="Component2">
        <service name="OneWayService">
          <!-- This service uses the @oneway annotation to specify this operation only has an input message
                and no output message.   -->
           <wm:workManager value="wm/scatest"/>
           <!-- This service specifies a workManager where the jndiName is wm/scatest. -->
        </service>
        <implementation.java class="test.Component2Impl"/>
        <reference name="component3" target="Component3"/>
      </component>
```

```
    <!-- component service with @oneway (non blocking operation -->
    <component name="Component3">
      <!-- By not defining the workManager, the SCADefaultWorkmanager that is created by the SCA
           runtime environment is used here. -->
      <implementation.java class="test.Component3Impl"/>
    </component>

</composite>
```

What to do next

Deploy the SCA component or application.

Configuring Java serialization for the SCA default binding

You can convert objects to a wire format, or *serialize* the objects, before transmitting them. You can serialize objects for applications that use the Service Component Architecture (SCA) default binding.

About this task

When using the default binding, objects are serialized to one of several wire formats before being sent. By default, objects are serialized to XML and sent as text over the wire. However, XML serialization occurs only with classes that are JAXB serializable. These classes require a default, no-argument constructor as well as getter and setter methods for each field.

To use non-JAXB serializable objects, you must specify a different wire format in the composite definition file. Specify the default binding element `wireFormat.javaObject` as a child element.

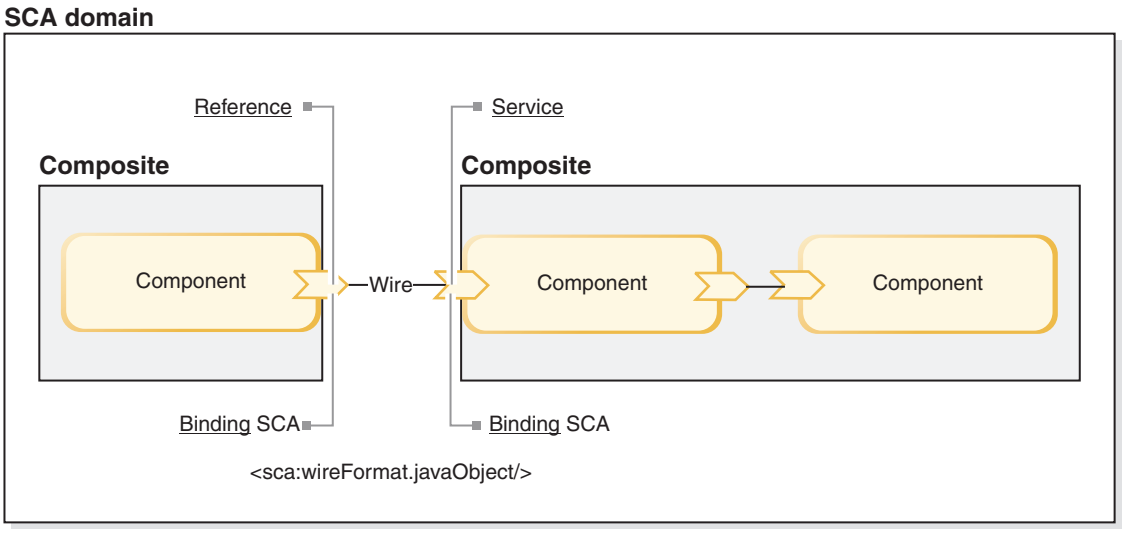


Figure 3. For non-JAXB serializable objects, use the wire format javaObject for the SCA default binding

When `wireFormat.javaObject` is used, objects use Java serialization instead of XML and travel as a stream of bytes over the wire. This enables you to use, for example, non-JAXB serializable Enterprise JavaBeans with the default binding.

Procedure

1. Open an editor on the composite definition file for your SCA application.
2. Add Java serialization in the default binding of the composite definition file.
   a. Add the product SCA namespace to the composite:

```
<composite xmlns:sca="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06">
```

b. For each service or reference `<binding.sca>` element, add a `wireFormat.javaObject` child element using that namespace:

```
<sca:wireFormat.javaObject/>
```

3. Save the changes to the composite definition file.

Results

You have configured Java serialization over the default bindings for your SCA service or reference.

Using the incorrect wire format might cause a runtime exception in your application.

Example

The following composite definition file configures Java serialization in the SCA default binding:

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      targetNamespace="http://scajavaserialize"
      xmlns:sca="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
      name="sca-java-serialize-backend">

  <component name="SCAJavaSerializationBackendComponent">
    <implementation.java class="test.bindings.sca.SerializeBackendImpl"/>

    <service name="SerializeBackendService">
      <interface.java
         interface="test.bindings.sca.SerializeBackendService"
         callbackInterface="test.bindings.sca.SerializeCallback">
      <binding.sca>
         <sca:wireFormat.javaObject/>
      </binding.sca>

      <callback>
         <binding.sca>
            <sca:wireFormat.javaObject/>
         </binding.sca>
      </callback>
    </service>

  </component>

</composite>
```

What to do next

Deploy your SCA component in an application.

Using the SCA default binding to find and locate SCA services

The product supports APIs that Service Component Architecture (SCA) and non-SCA clients can use to find and invoke SCA services over the SCA default binding.

About this task

A non-SCA client can use the OASIS SCAClientFactory API to obtain a service proxy for an OASIS SCA service in the same domain (product cell). An OASIS SCA client also can use this API as an alternative to wiring a reference in the composite file.

A non-SCA client can use the product CompositeContext API to obtain a service proxy for an OSOA SCA service in the same domain. An OSOA SCA client also can use this API as an alternative to wiring a reference in the composite file.

The SCA service must be deployed, running, and accessible over the default binding, `<binding.sca>`.

Procedure

- Obtain a service proxy for an OASIS SCA service in the same domain.

  The following example shows how to use the OASIS SCAClientFactory API:

  ```
  import org.oasisopen.sca.client.SCAClientFactory
  SCAClientFactory scaClientFactory = SCAClientFactory.newInstance(URI.create("default"));
  EchoService echoService = scaClientFactory.getService(EchoService.class, "SCASimpleEchoService");
  ```

- Obtain a service proxy for an OSOA SCA service in the same domain.

  The following example shows how to use the product CompositeContext API:

  ```
  import com.ibm.websphere.sca.context.CurrentCompositeContext;
  import com.ibm.websphere.sca.context.CompositeContext;
  CompositeContext compositeContext = CurrentCompositeContext.getContext();
  EchoService echoService =
    (EchoService) compositeContext.getService(EchoService.class, "SCASimpleEchoService");
  ```

What to do next

To improve performance, the caller can cache the service proxy that is returned by `SCAClientFactory.getService()` or `CompositeContext.getContext()`. Caching `echoService` in the example can avoid calls to the service registry for subsequent requests, resulting in better performance.

Configuring the SCA web service binding

You can expose a Service Component Architecture (SCA) application as a web service by configuring an SCA web service binding.

About this task

The web service binding enables SCA applications to expose services as web services to external clients and gives SCA components access to external web services. External clients that access SCA services exposed as web services might or might not be implemented as an SCA component. You can use the web service binding element `<binding.ws>` within either a component service or a component reference definition. When this binding is used with a component service, the web service binding enables clients to access a service that is offered by a particular component as a web service. If the web service binding is used with a component reference, components in an SCA composite can consume an external web service and access it like any other SCA component.

The product supports Web Services Description Language (WSDL) Version 1.1 definitions that also conform to the WS-I Basic Profile Version 1.1 and Simple SOAP Binding Profile 1.0 standards, and use the document literal style. For OASIS-level applications only, the rpc-literal style is also supported as long as it conforms to the above WS-I profiles. All these conditions are required for support.

An SCA web service binding is a WSDL-based binding; meaning that the binding either references an existing WSDL document or enables you to specify enough information to generate a WSDL document.

You can typically use the same policy set functionality, that you use for defining qualities of service on Java EE-based web services that are running in the application server, to define quality of service definitions on SCA clients and services that use the SCA web service binding. When the instructions for administratively configuring these policy set definitions are different for SCA-based web services than they are for Java EE-based web services, then special topics and instructions are provided in this information

center. If there are no special instructions for SCA, configure the corresponding quality of service on the SCA-based web service the same way that you would configure that quality of service for a Java EE-based web service.

Note: An SCA web service binding provides support for providing and consuming services using the SOAP Version 1.1 over HTTP and SOAP V1.2 over HTTP protocols.

Note: The product does not support the following functions:

- Java API for XML-Based Web Services (JAX-WS) handlers when using a SCA web service binding
- Message Transmission Optimization Mechanism (MTOM) or SOAP with Attachments (SwA) binary message optimizations

Use an SCA web service binding without implementing JAX-WS handlers. Do not use SwA binary message optimizations or MTOM optimizations for transferring binary data between SCA clients and services that use the SCA web services binding. Instead of implementing MTOM or SwA binary message optimizations to send binary data, use the base64Binary XML Schema Definition (XSD) encoding to embed the data within the SOAP message.

Procedure

1. Configure an SCA service with an SCA web service binding.

   Depending on whether you develop your SCA service using the top-down approach starting with an existing WSDL file or you develop your SCA service using the bottom-up approach starting with existing Java code, you might or might not have a WSDL file available. Also, the WSDL file might define only a portType or it might include a port definition as well.

   - For SCA applications that are developed top-down starting from a WSDL port, you must refer to the port definition in the existing WSDL file by adding a `<binding.ws>` element as a child of your `<service>` element. An example of the syntax for this step follows:

     ```
     <binding.ws wsdlElement="<port target Namespace>#wsdl.port(<service name attr>/<port name attr>)"/>
     ```

     The location attribute of the `<address>` element for the port is ignored by the runtime environment when determining the URL at which your service is invoked.

     The following WSDL file and composite definition illustrate this scenario:

     WSDL file

     ```
     <wsdl:definitions targetNamespace="http://www.ibm.com/" xmlns:tns="http://www.ibm.com/" ...>
          ....
         <wsdl:portType name="MyPortType ">
          ....
         <wsdl:binding name="MyBinding" type="tns:MyPortType">
          ....
         <wsdl:service name="MyService">
             <wsdl:port binding="tns:MyBinding" name="MyPort">
                 <wsdlsoap:address location=""/>
             </wsdl:port>
         </wsdl:service>
     ```

     Composite definition

     ```
     <composite...>
         <component name="MyComponent">
           <implementation.java class="test.MyCompImpl"/>
             <service name="GuessAndGreetWrapped">
                 <interface.wsdl interface="http://www.ibm.com/#wsdl.interface(MyPortType)" />
                 <binding.ws wsdlElement="http:// www.ibm.com/#wsdl.port(MyService/MyPort)" />
             </service>
         </component>
          ...
     </composite>
     ```

   - For SCA applications that are developed top-down starting from a WSDL portType, you must create an empty `<binding.ws>` element as the child of your `<service>` element. The empty `<binding.ws>` element directs the runtime environment to generate a port that corresponds to your WSDL portType definition. The generated port uses a SOAP 1.1 over HTTP WSDL binding.

The following WSDL file and composite definition illustrate this scenario:

WSDL file

```
<wsdl:definitions targetNamespace="http://www.ibm.com/" xmlns:tns="http://www.ibm.com/" ...>
      ....
     <wsdl:portType name="MyPortType ">
```

Composite definition

```
<composite...>
     <component name="MyComponent">
       <implementation.java class="test.MyCompImpl"/>
         <service name="GuessAndGreetWrapped">
             <interface.wsdl interface="http://www.ibm.com/#wsdl.interface(MyPortType)" />
             <binding.ws/>
         </service>
     </component>
      ...
</composite>
```

- For SCA applications that are developed bottom-up starting from existing Java code, you must create an empty `<binding.ws>` element as the child of your `<service>` element. The empty `<binding.ws>` element directs the runtime environment to generate a WSDL portType that corresponds to your Java interface, and a port with a SOAP 1.1 over HTTP WSDL binding.

  The following example demonstrates the `<binding.ws>` element for this scenario:

```
<composite...>
     <component name="MyComponent">
       <implementation.java class="test.MyCompImpl"/>
         <service name="GuessAndGreetWrapped">
             <binding.ws/>
         </service>
     </component>
      ...
</composite>
```

2. Determine the endpoint URL and test the endpoint of your deployed SCA service.

   After you configure the SCA service with a web service binding, you can test the endpoint for your deployed SCA service. Accessing the endpoint with a web browser displays the Axis2 cover page.

   The endpoint URL is determined based on one of the following scenarios:

   - If there is no `uri` attribute on the binding, the component resembles:

     ```
     <component name="C1">
       <service name="S1">
       <binding.ws/>
     ```

     Then the endpoint is:

     `http://<host>:<port>/C1_name/S1_name`

   - If the `uri` attribute on the binding is relative, the component resembles:

     ```
     <component name="C1">
       <service name="S1">
       <inding.ws name="B1" uri="U1"/>
     ```

     Then the endpoint is:

     `http://<host>:<port>/C1/U1`

   - If the `uri` attribute on the binding is absolute, which applies to OASIS only and not to OSOA, the component resembles:

     ```
     <component name="C1">
       <service name="S1">
       <inding.ws name="B1" uri="/U1"/>
     ```

     Then the endpoint is:

     `http://<host>:<port>/U1`

   For SCA services that are configured using `wsdlElement#wsdl.service`, the ports are appended; for example:

   `http://<host>:<port>/Component_name/Service_name/Port_name`

   For OASIS, `wsdl.service` is not supported.

3. View WSDL or XSD documents for the SCA service with the web service binding.

Append the query string ?wsdl to the endpoint URL. Accessing the URL with a web browser returns the outermost WSDL. You can then view included and imported documents using relative URLs.

4. Configure an SCA reference (client) with an SCA web service binding.
   a. Configure the reference with an <interface.wsdl> element that refers to the portType of the target service. Read about developing SCA service clients to learn how to configure the reference with an <interface.wsdl>.
   b. Resolve the SCA reference to an actual endpoint of a deployed web service using one of the mechanisms provided by the SCA web service binding support.
      - When the target web service is deployed as an SCA component service in the same domain as the client component, you can resolve the reference to the target component using the @target attribute of the <reference> element for OSOA composites or the <binding.ws> element for OASIS composites. Using the @target attribute eliminates the need to know the specific URL of the deployed target service.

        For example, assume the target service is defined as follows:

        ```
        <component name="TargetComponent">
          <implementation.java .../>
          <service name="MyService">
              <interface.wsdl ... />
              <binding.ws ... />
          </service>
        </component>
        ```

        A reference in an OSOA composite can be wired to this service as follows:

        ```
        <component name="ClientComponent">
          <implementation.java .../>

            <!-- Resolution uses @target attribute on reference element. -->
            <reference name="myRef" target="TargetComponent/MyService">
               <interface.wsdl ... />

               <!-- The binding does not need endpoint-related info added. -->

               <binding.ws/>
            </reference>
        </component>
        ```

        A reference in an OASIS composite can be wired to this service as follows:

        ```
        <component name="ClientComponent">
          <implementation.java .../>

            <reference name="myRef">
               <interface.wsdl ... />

               <!-- Resolution done using the @target attribute on binding element. -->
               <binding.ws target="TargetComponent/MyService"/>
            </reference>
        </component>
        ```

        In these examples, the binding.ws element does not specify a wsdlElement attribute. The SCA run time creates a SOAP binding, WSDL service, and WSDL port on each side. If your service binding specifies a SOAP binding, either directly or indirectly using the WSDL port, your reference binding must specify a compatible SOAP binding.
      - If the target web service is not an SCA service in the same domain as the client, you must use a binding-specific endpoint resolution mechanism. You can also resolve a reference to an SCA service in the same domain by using the binding-specific mechanisms, instead of using the @target annotation.
        1) You can define the endpoint for a deployed web service in an existing WSDL file using <wsdlsoap:address>. For example:

           ```
           <wsdl:definitions targetNamespace="http://my.work/test/"
                   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
               xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" ... >
               ...
               <wsdl:portType name="MyPortType">
               ...
               <wsdl:binding name="MyBinding" type="tns:MyPortType">
           ```

```
    ...
    <wsdl:service name="MyService">
      <wsdl:port name="MyPort" binding="tns:MyBinding">
        <wsdlsoap:address
           location="http://www.mywork.com:9080/TargetComponent/MyService "/>
     </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

The client points to the WSDL port using a @wsdlElement attribute on the <binding.ws> element using the following syntax:

```
<port target Namespace>#wsdl.port(service_name_attribute/port_name_attribute)"/>
```

For example:

```
<component name="ClientComponent">
  <implementation.java .../>
    <reference name="myRef">
       <interface.wsdl ... />
       <binding.ws wsdlElement="http://my.work/test/#wsdl.port(MyService/MyPort)"/>
    </reference>
</component>
```

2) If the endpoint is not present in the WSDL for the service, there are several ways to specify the endpoint using the reference. In the following WSDL file, the endpoint (wsdlsoap:*address*) is not specified.

```
<wsdl:definitions targetNamespace="http://my.work/test/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"   ... >
  ...
  <wsdl:portType name="MyPortType">
  ...
  <wsdl:binding name="MyBinding" type="tns:MyPortType">
  ...
    <wsdl:service name="MyService">
      <wsdl:port name="MyPort" binding="tns:MyBinding">
      ...
        <!-- In this case, the endpoint is not specified. -->
        <wsdlsoap:address location=""/>
     </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

For OSOA, you can add the endpoint to the composite definition. Add the @uri attribute to the <binding.ws> element to specify the endpoint. For example:

```
<!-- An OSOA example composite definition. -->
<component name="ClientComponent">
  <implementation.java .../>
    <reference name="myRef">
         <interface.wsdl ... />
         <binding.ws wsdlElement="http://my.work/test/#wsdl.port(MyService/MyPort)"
            uri="http://www.mywork.com:9080/TargetComponent/MyService" />
    </reference>
</component>
```

For OASIS based references, one approach is to add a WSDL binding to the reference. Then add @uri to the component <binding.ws> element.

```
<!-- An OASIS example composite definition. -->
<component name="ClientComponent">
  <implementation.java .../>
    <reference name="myRef">
         <interface.wsdl interface="http://my.work/test/#wsdl.interface(MyPortType)"/>
         <binding.ws uri="http://www.mywork.com:9080/TargetComponent/MyService"
                     wsdlElement="http://my.work/test/#wsdl.binding(MyBinding)" />
    </reference>
</component>
```

5. Optional: If your service or reference interface is bidirectional such that a callback is defined, you must also configure the web service binding on the callback.
   - Configure callback for your SCA web services binding using the WSDL port.

     Configuring callback for your SCA web services binding is similar to configuring an SCA service with the SCA web services binding; however, you add a second service, the callback. When you configure an SCA service with the SCA web services binding, you define a WSDL port, either explicitly by pointing directly to a WSDL port definition, or implicitly by giving the runtime enough information to calculate a WSDL port. Similarly, you must also define a WSDL port for the callback.

A difference for the callback is that the runtime environment defines the WSDL port that is used for the callback because the runtime environment must keep this port tightly coupled to the forward call. Therefore, the most you can do when developing your SCA application using the top-down approach and defining a callback for this service, is to point to a WSDL binding. For example:

```
<!-- Configuring a service with callback with web service binding -->
 <component name="HelloWorldServiceComponent">
   <implementation.java class="..." />
   <service name="HelloWorldService">
     <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                     callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
     <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.port(HelloWorldService/HelloWorldSoapPort)"/>
       <callback>
         <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.binding(HelloWorldCallbackSoapBinding)"/>
       </callback>
    </service>
 </component>
```

The following example shows a configuration of a client component with reference to this service with callback defined. The reference and service configuration each share the same view of which direction is the forward direction and which is the callback direction.

```
<!-- Configuring a reference with callback with web service binding. -->
 <component name="HelloWorldClientComponent">
   <implementation.java class="..." />
   <reference name="helloWorldRef">
     <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                     callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
     <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.port(HelloWorldService/HelloWorldSoapPort)"/>
       <callback>
         <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.binding(HelloWorldCallbackSoapBinding)" />
       </callback>
   </reference>
 </component>
```

- Configure callback for your SCA web service binding using the WSDL portType.

  Similar to the scenario of configuring a service with a web service binding when starting with a WSDL port, you also configure an empty <binding.ws> element to configure callback using the WSDL portType. The following composite definition example illustrates the scenario when starting with two WSDL portType definitions that has such that one interface uses a forward direction and the other interface uses callback:

```
<!-- Configuring a service with callback with web service binding -->
 <component name="HelloWorldServiceComponent">
   <implementation.java class="..." />
   <service name="HelloWorldService">
     <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                     callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
     <binding.ws/>
        <callback>
           <binding.ws/>
        </callback>
   </service>
 </component>
```

```
<!-- Configuring a reference with callback with web service binding -->
 <component name="HelloWorldClientComponent">
   <implementation.java class="..." />
   <reference name="helloWorldRef">
     <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                     callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
     <binding.ws/>
       <callback>
         <binding.ws/>
       </callback>
    </reference>
 </component>
```

- Configure callback for your SCA web service binding using the Java interface.

  For the bottom-up case starting with a Java interface, the composite definition is identical to the WSDL port and portType scenarios, except that you must replace the <interface.wsdl> elements with the <interface.java> element. For example:

```
<interface.java interface="helloworld.HelloWorldService"
        callbackInterface="helloworld.HelloWorldCallback"/>
```

- (Optional: OASIS only) Configure your service to use SOAP 1.1 or 1.2.

  You can configure your SCA web service to require a particular SOAP version by providing an intent. For example:

  ```
  <binding.ws requires="sca:SOAP.v1_2"/>
  ```

  Supported intents are `sca:SOAP.v1_1` and `sca:SOAP.v1_2`.

Results

You have a configured SCA web service binding service or reference.

There are additional ways for the web service binding to generate a WSDL port that are not described in this topic. These additional methods rely on WSDL generation at run time. This can cause problems if the generated WSDL does not match the original WSDL obtained from a service provider.

You can avoid problems by ensuring that client package references the original WSDL obtained from a web service provider. If you use the shortcut of omitting a client-side reference to the WSDL, be sure to do so only in the case when you are sure the WSDL port that is generated for the client is identical to the WSDL port of the deployed service because the service port is generated using the same algorithm.

For an example that is not problematic, suppose you write a service using the bottom-up style, starting from a Java interface, and deploy the service with a `<binding.ws>` element with no attributes. This directs the runtime environment to generate the WSDL port for this service. Also suppose an SCA client is developed with access to the original Java classes used to write the service implementation. This SCA client is used to test the SCA service using a client-side reference with web service binding. You can configure this reference without any knowledge of the service WSDL. In this case, the reference interface is the original Java interface of the service, and you can resolve the reference using the `<reference>` @target mechanism. See the resolving SCA references documentation for more information about using the @target attribute to resolve an SCA reference. Using this approach, there is no WSDL to obtain or refer to in constructing the client. This works because the product runtime environment maps the service-side Java to WSDL in an identical manner as it maps the client-side Java to WSDL.

In contrast, the following scenario is problematic. Suppose that you write an SCA client with a web service binding reference to a web service that is hosted on a platform other than the product. It might seem reasonable to generate your Java client from the service provider, and then ignore the WSDL from that point on, avoiding the additional syntax in your client-side composite definition. To do this, you use the `<binding.ws>` element @uri attribute to specify the endpoint URL where the service is hosted. This scenario is problematic because it forces the runtime environment to generate a WSDL port for the client which might result in subtle mismatches between the WSDL generated for the client side and the actual WSDL port description of the deployed web service.

What to do next

Deploy your composite that has the SCA web services binding service or reference.

Note: If the composite uses a bidirectional interface as an SCA service interface and a NullPointerException results when you add your SCA module or application to a business-level application (for example, using the wsadmin AdminTask.addCompUnit command), you might need to add the SCA @Callback annotation to the forward Java interface.

A NullPointerException can occur whether you are using a top-down development style (starting from WSDL or XSD) or a bottom-up style (starting from Java). With the top-down style, a composite definition typically includes a bidirectional interface definition such as the following:

```
<interface.wsdl interface="http://forward.my/intf#wsdl.interface(ForwardIntf)"
    callbackInterface="http://callback.my/intf#wsdl.interface(CallbackIntf)"/>
```

With the bottom-up style, a composite definition typically includes a bidirectional interface definition such as the following:

```
<interface.java interface="my.forward.intf.ForwardIntf"
    callbackInterface="my.callback.intf.CallbackIntf"/>
```

The NullPointerException can occur if the forward Java interface is not annotated with @Callback, or more specifically @org.osoa.sca.annotations.Callback. In the top-down example, the forward Java interface is the class generated from portType, `http://forward.my/intf/ForwardIntf` using the wsimport tool or `my.forward.intf.ForwardIntf` if the default options are used. In the bottom-up example, the forward Java interface is the class `my.forward.intf.ForwardIntf`.

To fix the NullPointerException problem, add the @Callback annotation to the forward Java interface and recompile. The @Callback annotation is a class-level annotation with a single argument, the callback interface Class object; for example:

```
...
import org.osoa.sca.annotations.Callback
@Callback(my.callback.intf.CallbackIntf.class)
public interface ForwardIntf  {
...
```

If you are using the recommended top-down development style, the forward Java interface is a generated class. It is generated from the WSDL portType using the wsimport tool. Use the `-s` or `-keep` option so that wsimport generates the Java source files, and then add the @Callback annotation manually to the generated class and recompile.

Configuring web service binding custom endpoints to support a proxy server

You can configure custom service endpoints for Service Component Architecture (SCA) web service bindings that are accessed by Hypertext Transfer Protocol (HTTP).

Before you begin

Before you begin this task, install your SCA application.

About this task

When a service is exposed over the SCA web service binding, the service endpoint is specific to the server in which the service is hosted. Clients use this endpoint URI to access the service. In some cases, you may want clients to indirectly reference the service by using a proxy server as the service endpoint. For example, a proxy server is required to implement clustered web service binding endpoints. To enable clients to use a proxied endpoints, there are two ways to do this:

- If your endpoints are specified in the SCA contributions composite definition or WSDL document location attribute, you must specify the proxy server endpoint instead of the WebSphere server specific endpoint.
- If your client resolves the endpoint by using the `target` attribute in your client composite definition, use the administrative console to configure the custom endpoints for SCA composites that are accessed by the Hypertext Transfer Protocol (HTTP) protocol. This approach is the most flexible for SCA clients within the same domain as their service providers. When using the `target` attribute, SCA references can resolve the service endpoints without the client specifying endpoints in the composite definition or WSDL document.

Procedure

1. In the administrative console, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **Provide HTTP endpoint URL information**.
2. Select the HTTP endpoint URL prefix.

   When entering custom endpoints, specify one and only one endpoint URL prefix each for the HTTP and HTTPS protocols.

Results

You have configured web service bindings custom endpoints.

What to do next

You can configure the bindings to do transport layer authentication.

Configuring the SCA web service binding to transport SOAP messages over JMS

You can configure a Service Component Architecture (SCA) web service binding to transport SOAP messages over Java Message Service (JMS) protocol.

Before you begin

For information about web service bindings, read “Configuring the SCA web service binding” on page 747.

About this task

Web services technology plays an important role in most service-oriented architecture (SOA) solutions relevant today, including SCA. The web service binding type enables SCA applications to expose services as web services to external clients and gives SCA components access to external web services. Using JMS as a transport for web services provides a reliable asynchronous messaging transport for request and response.

This topic describes how to configure a web service binding to flow SOAP messages over JMS.

Note: The web service binding supports Web Services Description Language (WSDL) Version 1.1. Further, the web service binding supports industry standard SOAP/JMS protocol and IBM proprietary SOAP over JMS protocol.

Procedure

1. Identify and configure JMS resources using the administrative console or the wsadmin scripting tool.

   Refer to topics on creating JMS resources based on different JMS providers.

   For example, to identify and configure JMS resources based on the default JMS provider, do the following:

   a. Create a service integration bus and associate a bus member.
   b. Create destinations for the request and response for the service integration bus.
   c. Create destinations for the request and response under JMS resources.
   d. Create an activation specification that is associated with a request destination.
   e. Create connection factories to process the request and to send the response.
2. Configure an SCA service with an SCA web service binding to transport SOAP messages over JMS.

   Configure a web service binding as described in “Configuring the SCA web service binding” on page 747 with three additional attributes under the `<binding.ws>` element to use a JMS transport:

   - Web service binding endpoint `uri` attribute
   - `@activationSpec` attribute
   - Response `@responseConnectionFactory` attribute

   a. Open an editor on the SCA composite definition file that specifies an SCA service with a web service binding.
   b. In the composite definition file, configure the web service binding endpoint `uri` attribute.

Specify the uniform resource identifier (URI) in SOAP over JMS endpoint uniform resource locator (URL) syntax. A JMS endpoint URL accesses web services with a JMS transport. The URL specifies the JMS destinations, connection factory, and port component name for the web service request.

Specify the URI in syntax that supports either of the following protocols:

- Industry standard SOAP/JMS protocol (recommended)
- IBM proprietary SOAP over JMS protocol (deprecated)

URI based on industry standard SOAP over JMS protocol

The syntax for a URI based on industry standard SOAP over JMS protocol is as follows:

```
uri=jms:jndi:<destination_JNDI_name>?<property>=<value>&<property>=<value>& ...
```

The URL consists of the `jms:` transport type, followed by the `jndi:` variant type, followed by the Java Naming and Directory Interface (JNDI) name of the destination queue or topic, followed by the query string containing a list of property and value pairs that specify JMS endpoint information. The `jndi:` variant means that JNDI is used to locate object names in the endpoint URL string.

For *property*, you can specify URL properties such as the following:

Table 85. JMS endpoint URL properties typically used for a URI based on industry standard SOAP over JMS protocol. The properties are jndiConnectionFactoryName, targetService, and replyToName.

Property name	Description
jndiConnectionFactoryName	Specifies the JNDI name of the connection factory that is used by the client run time to establish a connection to the JMS messaging engine. jndiConnectionFactoryName is optional for the service `uri` attribute.
targetService	Specifies the target service to which to deliver the message. For a forward call from a service to a reference, targetService has the format `serviceComponentName/serviceName`. In a callback `uri`, targetService has the format `referenceComponentName/referenceName`.
replyToName	Specifies the JNDI name of the JMS destination to which the response message is sent. replyToName is a JMS-related property that enables the client to use a previously defined, permanent queue rather than a temporary queue, for receiving replies.

For a complete list of properties supported, refer to the topic on JMS endpoint URL syntax.

A URI based on industry standard SOAP over JMS protocol resembles the following:

```
uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
 =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
 &amp;targetService=AccountServiceComponent/AccountService"
```

URI based on IBM proprietary SOAP over JMS protocol

Attention: IBM proprietary SOAP over JMS protocol is deprecated.

The syntax for a URI based on IBM proprietary SOAP over JMS protocol is as follows:

```
uri=jms:/[queue|topic]?<property>=<value>&<property>=<value>& ...
```

The URL consists of the `jms:` transport type, followed by either `/queue` or `/topic` to specify the JMS destination type, followed by the query string containing a list of property and value pairs that specify the JMS endpoint information.

For *property*, you can specify URL properties such as the following:

Table 86. JMS endpoint URL properties typically used for a URI based on IBM proprietary SOAP over JMS protocol. The properties are destination, connectionFactory, replyToDestination, and targetService.

Property name	Description
destination	Specifies the JNDI name of the destination queue or topic.
connectionFactory	Specifies the JNDI name of the connection factory. connectionFactory is optional for the service `uri` attribute.
replyToDestination	Specifies the JNDI name to which a response is sent. replyToDestination is optional for the service `uri` attribute.
targetService	Specifies the target service to which to deliver the message. For a forward call from a service to a reference, targetService has the format `serviceComponentName/serviceName`. In a callback `uri`, targetService has the format `referenceComponentName/referenceName`.

For a complete list of properties supported, refer to the IBM proprietary SOAP over JMS protocol topic.

A URI based on IBM proprietary SOAP over JMS protocol resembles the following:

```
uri="jms:/queue?destination=jms/MyBankAccountService_Request&amp;connectionFactory
 =jms/MyBankAccountService_CF&amp;replyToDestination=jms/MyBankAccountService_Response
 &amp;targetService=AccountServiceComponent/AccountService"
```

c. In the composite definition file, configure an `@activationSpec` attribute under the `<binding.ws>` element.

   The attribute identifies the activation specification that the service uses to connect to the JMS destination and to process incoming request messages.

   For the destination associated with this activation specification, specify the request destination defined in the web service binding endpoint `uri`; for example:

   ```
   soapjms:activationSpec="jms/AccountActivationSpec"
   ```

   Qualify the OSOA SCA composite by setting `xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"`.

   Qualify the OASIS SCA composite by setting `xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"`.

d. In the composite definition file, configure a response `@responseConnectionFactory` attribute under the `<binding.ws>` element to send a response back to the client when request-response messaging pattern is used.

   For example:

   ```
   soapjms:responseConnectionFactory="jms/Account_Response_CF"
   ```

   Qualify the OSOA SCA composite by setting `xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"`.

   Qualify the OASIS SCA composite by setting `xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"`.

After these updates are made to the `<binding.ws>` element, the OSOA SCA composite definition file resembles:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
 xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
 targetNamespace="http://www.ibm.com/samples/sca/mybank"
 name="MyBank">
  <component name="AccountServiceComponent">
    <implementation.java
      class="samples.mybank.AccountServiceImpl"/>
    <service name="AccountService">
      <interface.wsdl
      interface=
          "http://www.mybank.com/account#wsdl.interface(AccountService)" />
      <binding.ws
          wsdlElement=
            "http://www.mybank.com/account#wsdl.port(AccountService/AccountServicePort)"

          uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
              =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
              &amp;targetService=AccountServiceComponent/AccountService"

          soapjms:activationSpec="jms/MyBankAccountService_AS"

          soapjms:responseConnectionFactory="jms/MyBankAccountService_CF"/>
    </service>
  </component>
</composite>
```

The OASIS SCA composite definition file is similar except for the namespace prefixes, which are:

```
xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"
```

3. Configure an SCA reference (client) with an SCA web service binding to transport SOAP message over JMS.

   Configure a web service binding with a web service binding endpoint `uri` attribute specific to JMS transport under the `<binding.ws>` element.

a. Open an editor on the SCA composite definition file that specifies an SCA reference with a web service binding.
b. In the composite definition file, configure the web service binding endpoint `uri`.

   Specify the URI in SOAP over JMS endpoint URL syntax. The URL specifies the JMS destinations, connection factory, and port component name for the web service request. A JMS endpoint URL can be based on industry standard SOAP/JMS protocol (recommended) or on IBM proprietary SOAP over JMS protocol. For a description of the URI syntax, see step 2(b). In SOAP over JMS support for web service bindings, you must use the `<binding.ws>` `uri` attribute for endpoint resolution to the target component; you cannot use the SCA reference `@target` attribute.

   The URI resembles the following:

```
uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
 =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
 &amp;targetService=AccountServiceComponent/AccountService"
```

After you define the URI in the `<binding.ws>` element, the OSOA SCA composite definition file resembles:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
targetNamespace="http://www.ibm.com/samples/sca/mybank"
name="MyBankClient">
  <component name="AccountSummaryService">
    <implementation.java
      class="samples.mybank.AccountSummaryServiceImpl"/>
    <reference name="accountService">
    <interface.wsdl
      interface=
        "http://www.mybank.com/account#wsdl.interface(AccountService)" />
    <binding.ws
      wsdlElement=
        "http://www.mybank.com/account#wsdl.port(AccountService/AccountServicePort)"

       uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
            =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
            &amp;targetService=AccountServiceComponent/AccountService"/>
    </reference>
  </component>
</composite>
```

In an OASIS SCA composite definition file, the binding element cannot include both the `uri` and `wsdlElement` attributes. You must use one or the other. If you use `wsdlElement`, the WSDL port definition in the WSDL file specifies the URI as shown in the following example:

```
<wsdl:service name="AccountService">
  <wsdl:port binding="account:AccountServiceSOAP" name="AccountServicePort">
    <soap:address
      location="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
        =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
        &amp;targetService=AccountServiceComponent/AccountService"/>
  </wsdl:port>
</wsdl:service>
```

Results

You have a configured an SCA web service binding service or reference.

Example

Configuring a request-response Message Exchange Pattern for SOAP over JMS

This example describes a `<binding.ws>` element in the composite definition file for a request-response message exchange pattern from an SCA component reference to an SCA component service.

The following example shows an OSOA SCA component with a reference binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
           targetNamespace="http://www.ibm.com/samples/sca/mybank" name="MyBankClient">
  <component name="AccountSummaryService">
    <implementation.java class="samples.mybank.AccountSummaryServiceImpl"/>
    <reference name="accountService">
      <interface.wsdl interface="http://www.mybank.com/account#wsdl.interface(AccountService)"/>
      <binding.ws
        wsdlElement="http://www.mybank.com/account#wsdl.port(AccountService/AccountServicePort)"
        uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
          =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
          &amp;targetService=AccountServiceComponent/AccountService"/>
    </reference>
  </component>
</composite>
```

The following example shows an OSOA SCA component with a service binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
           targetNamespace="http://www.ibm.com/samples/sca/mybank" name="MyBank">
  <component name="AccountServiceComponent">
    <implementation.java class="samples.mybank.AccountServiceImpl"/>
    <service name="AccountService">
      <interface.wsdl interface="http://www.mybank.com/account#wsdl.interface(AccountService)"/>
        <binding.ws
          wsdlElement="http://www.mybank.com/account#wsdl.port(AccountService/AccountServicePort)"
          uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
            =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response
            &amp;targetService=AccountServiceComponent/AccountService"
          soapjms:activationSpec="jms/MyBankAccountService_AS"
          soapjms:responseConnectionFactory="jms/MyBankAccountService_CF"/>
    </service>
  </component>
</composite>
```

You can use these composite definition examples for @OneWay Message Exchange Pattern. When using @OneWay, the reference binding URI does not need to specify `replyToName` because there is no reply for the @OneWay call.

Configuring an SCA callback for SOAP over JMS

This example describes a `<binding.ws>` element in the composite definition file for a callback message exchange pattern between an SCA component reference and an SCA component service.

The following example shows an OSOA SCA component with a reference binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
           targetNamespace="http://www.ibm.com/samples/sca/mybank" name="MyBankClient">
  <component name="AccountSummaryService">
    <implementation.java class="samples.mybank.AccountSummaryServiceImpl"/>
    <reference name="accountService">
      <interface.wsdl
         interface="http://www.mybank.com/account#wsdl.interface(AccountService)"
         callbackInterface="http://www.mybank.com/account#wsdl.interface(AccountServiceCallback)"/>
      <binding.ws
         wsdlElement="http://www.mybank.com/account#wsdl.port(AccountService/AccountServicePort)"
         uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
            =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response&amp;
            targetService=AccountServiceComponent/AccountService"/>
        <callback>
          <binding.ws
             wsdlElement="http://www.mybank.com/account#wsdl.binding(AccountServiceCallback)"
             uri="jms:jndi:jms/MyBankAccountService_Callback?jndiConnectionFactoryName
              =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response&amp;
```

```
            targetService=AccountServiceComponent/AccountService"
           soapjms:activationSpec="jms/$MyBankAccountService_Callback_AS"
           soapjms:responseConnectionFactory="jms/MyBankAccountService_CF"/>
        </callback>
    </reference>
  </component>
</composite>
```

For the OASIS SCA composite, the `soapjms` namespace is:

```
xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"
```

`targetService` is the target service to which a message is delivered. In a forward call from a service to a reference, specify the targetService in the format *serviceComponentName/serviceName*. In the callback URI, specify the targetService in the format *referenceComponentName/referenceName*.

The callback binding on the reference becomes an SCA service binding for the callback call. Hence activationSpec and responseConnectionFactory must be defined.

The following example shows an OSOA SCA component with a service binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
           targetNamespace="http://www.ibm.com/samples/sca/mybank" name="MyBank">
  <component name="AccountServiceComponent">
    <implementation.java class="samples.mybank.AccountServiceImpl"/>
    <service name="AccountService">
      <interface.wsdl interface="http://www.mybank.com/account#wsdl.interface(AccountService)"
           callbackInterface="http://www.mybank.com/account#wsdl.interface(AccountServiceCallback)"/>
        <binding.ws
           wsdlElement="http://www.mybank.com/account#wsdl.port(AccountService/AccountServicePort)"
           uri="jms:jndi:jms/MyBankAccountService_Request?jndiConnectionFactoryName
            =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response&amp;
            targetService=AccountServiceComponent/AccountService"
           soapjms:activationSpec="jms/MyBankAccountService_AS"
           soapjms:responseConnectionFactory="jms/MyBankAccountService_CF" />
          <callback>
            <binding.ws
               wsdlElement="http://www.mybank.com/account#wsdl.binding(AccountServiceCallback)"
               uri="jms:jndi:jms/MyBankAccountService_Callback?jndiConnectionFactoryName
                =jms/MyBankAccountService_CF&amp;replyToName=jms/MyBankAccountService_Response&amp;
                targetService=AccountServiceComponent/AccountService" />
          </callback>
    </service>
  </component>
</composite>
```

For the OASIS SCA composite, the `soapjms` namespace is:

```
xmlns:soapjms="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"
```

The service binding callback element does not need to specify a JMS endpoint URI because the callback URI is derived from the reference binding.

The callback binding on the service becomes a SCA reference binding for the callback call from the service back to the reference. Hence activationSpec and responseConnectionFactory need not be defined.

In some examples, the `uri` value is shown on multiple lines for publication. In your composite definition, place the `uri` value on one line.

What to do next

Deploy the SCA composite in an application and test the flow of messages over the web service binding.

Configuring EJB bindings in SCA applications

Use this task to learn how to use Enterprise JavaBeans (EJB) bindings in SCA applications.

Support is provided for EJB bindings in 2.x and 3.x-style beans, for service and reference.

The EJB bindings do not support `interface.wsdl` files.

The following is an example of an composite definition that has a service exposed over an EJB 3.x binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://neworder/sca/jdbc"
name="NewOrderComposite">

<component name="NewOrderEJB3ServiceComponent">
<implementation.java class="neworder.sca.jdbc.NewOrderServiceImpl" requires="managedTransaction.local"/>
<service name="NewOrderService" requires="suspendsTransaction">
<interface.java interface="neworder.sca.jdbc.NewOrderService"/>
<binding.ejb ejb-version="EJB3"/>
</service>
</component>
</composite>
```

A client that wants to invoke the resultant enterprise bean would treat it like any other enterprise bean and not like a regular SCA service. CompositeContext.getService is not supported for a non-SCA binding, therefore, a getService() on the CompositeContext would not work here. The following is the client code for the previous example:

```
InitialContext ctxt = new InitialContext();
  Object remoteObj = ctxt.lookup("ejb/sca/ejbbinding/NewOrderEJB3ServiceComponent/NewOrderService#neworder.sca.jdbc.NewOrderServiceRemote");
  NewOrderServiceRemote newOrderRemote = (NewOrderServiceRemote) PortableRemoteObject.narrow(remoteObj, NewOrderServiceRemote.class);
```

The following is an example of an composite definition that contains references to both EJB 2.x and EJB 3.x bindings:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
targetNamespace="http://erww.workload" name="ConvertComposite">

<component name="ConvertInputOutputServiceComponent">
<implementation.java class="convert.inputoutput.sca.ConvertInputOutputServiceImpl"

<reference name="priceQuoteSessionReference">
<interface.java interface="priceQuoteSession.PriceQuoteSession"/>
<binding.ejb uri="corbaname:iiop:localhost:2809/NameServiceServerRoot#ejb/session/PriceQuoteSessionFacadeBean"/>
</reference>
</component>
</composite>
```

Different binding.ejb attributes can be used for service side EJB bindings or reference side EJB bindings. The following information explains how the default value is calculated for each side:

Service side

The service side EJB binding applies only to Java archive (JAR)-packaged SCA applications.

EJB 2.0-level beans

URI is the JNDI name for the home; it can be calculated with the default short name in the following form:

`/sca/ejbbinding/`*`component_name`*`/`*`service_name`*

Therefore, the URI can be calculated as:

`corbaname:iiop:localhost:2812/NameServiceServerRoot#ejb/sca/ejbbinding/`*`component_name`*`/`*`service_name`*

You can use it to look up home.

EJB 3.x-level beans

The URI contains the component-id, therefore, it is calculated the same as the EJB 2.0 beans as follows:

```
sca/ejbbinding/component_name/service_name
```

The URI can be calculated as:

```
corbaname:iiop:localhost:2812/NameServiceServerRoot#ejb/sca/ejbbinding/component_name/service_name#package.qualified.interface of
SCA Java interface with prefix of Remote or Local to the class name
```

You can use it directly to get the business interface.

The following code example displays as if it were a session bean:

```
<session name="ServiceNameBean" component-id="sca/ejbbinding/component_name/service_name"/>
```

When an SCA service is exposed through an EJB service binding, the service is exposed through an enterprise bean. During deployment, the SCA runtime generates a session bean for the service exposed through the EJB binding. The caller of the composite service can invoke this service by accessing the generated enterprise bean as if they are invoking any enterprise bean.

The generated enterprise bean for the composite service is in the *profile_root*/installedApps/*cell_name*/*sca.composite.nameApp.ear*/ directory. Callers need to include the client required classes, such as remote or home, of the generated bean in the classpath or bundle the classes in the JAR file. In addition, if the caller application is running on a non-WebSphere application server, a "Bare" Java Standard Edition (SE), or a version of WebSphere Application Server previous to version 7.0 without the Feature Pack for EJB 3.0, then you must run the **createEJBStubs** command for the generated EJB module to generate client-side stubs and include the generated subs at the client application. The **createEJBStubs** command is under the *profile_root*/installedApps/*cell_name*/*sca.composite.nameApp.ear*/ directory. See the topic on the create stubs command.

Reference side

The reference side EJB binding applies to both JAR-packaged applications and web application archive (WAR)-packaged applications, if not otherwise stated.

- The URI is used to lookup either the EJB 2.x home or EJB 3.x business interface. Follow the naming convention of the Java Enterprise Edition (JEE) specification if you are using an existing JEE EJB module. If you use an SCA service with the binding.ejb attribute, use the value mentioned above. For more information about the EJB 3.x JNDI name, see the topic EJB 3.x bindings overview.
- homeInterface: Not used
- ejb-link-name: Only applies to WAR-packaged SCA applications. When URI is not defined, use it to look up an EJB module that is defined in the same enterprise archive (EAR) as the WAR.
- session-type: default value "stateless"
- ejb-version: default value "EJB2"

Attention: A lookup issue for EJB 3.x reference bindings might occur when the URI follows the corbaname:iiop:host:port/NameServiceServerRoot##*ejb3_binding_longform* pattern This problem exists only for EJB 3.x reference bindings. When the EJB 3.x reference binding URI follows the corbaname:iiop:host:port/NameServiceServerRoot##*ejb3_binding_longform* pattern, where *ejb_binding_longform* is ejb/<component-id#<package.qualified.interface>, and if more than one enterprise bean that is implementing the same interface is deployed on that server, lookup may not be directed to the correct EJB with corresponding component ID.

An example of a URI where this problem can occur is as follows:

```
uri="corbaname:iiop:host:port/NameServiceServerRoot#ejb/EJB3CounterSample/EJB3Beans.jar
/StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter
```

There are two enterprise beans implementing the com.ibm.websphere.ejb3sample.counter.RemoteCounter interface. To avoid this issue:

- Use a URI that does not start with "corbaname:"
- Use a binding name in the URI that is an EJB binding short form, for example, corbaname:iiop:host:port/NameServiceServerRoot#<package.qualified.interface>.
- Use a binding name in the URI that is a unique user-defined binding name.
- Ensure that the two enterprise beans that are deployed on the server do not implement the same interface.
- Ensure that the EJB binding URI is pointing to an EJB 2.0 bean.

To resolve this problem, follow these guidelines:

- If the EJB reference binding is accessing an enterprise bean that is located in the same cell, the URI should not start with "corbaname:."
- For same cell lookup, the URI pattern should be one of the following.

  ```
  uri="ejb/EJB3CounterSample/EJB3Beans.jar/StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter"
  ```

  or

  ```
  uri="cell/clusters/cluster_name/ejb/EJB3CounterSample/EJB3Beans.jar
  /StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter"
  ```

  or

  ```
  uri="cell/nodes/node_name/servers/server_name//ejb/EJB3CounterSample/EJB3Beans.jar
  /StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter"
  ```
- Even in cross cell access, the recommended method is to create a namespace binding for the enterprise bean that is accessed by the EJB reference binding. After the namespace binding is created, use the namespace binding in the URI of the EJB reference binding as uri="cell/persistent/*name_in_namespace_binding*".

| Different patterns of the SCA EJB reference binding URI are based on the user setup and configurations.
| If the SCA EJB reference binding is accessing a stateless session bean on the same server, the EJB
| reference binding URI can be designated as the JNDI name, uri="ejb/com/app/resumebank/
| ResumeBankHome". If the SCA EJB reference binding is referencing another SCA service with an EJB
| binding in the same server, the URI can be designated as the JNDI name, uri="ejb/com/app/resumebank/
| ResumeBankHome".

If the EJB reference binding is accessing a stateless session bean that is deployed in the same cell, the URI can be based on cluster/node/server setup, for example:

```
uri="cell/clusters/cluster2/ejb/com/app/resumebank/ResumeBankHome"
uri="cell/nodes/node_name/servers/server_name/ejb/com/app/resumebank/ResumeBankHome"
```

If the EJB reference binding is accessing a stateless session bean on a different cell (cross cell) or a mixed cell, you need to create a namespace binding, either an enterprise bean or Corba type, in the administrative console and use the name in namespace binding in EJB reference binding URI such as, uri="cell/persistent/*name_in_namespace_binding*". For example, uri="cell/persistent/*neworder*" where *neworder* is name in the namespace binding.

Using EJB bindings in SCA applications in a cluster environment

Use this task to learn how to use Enterprise JavaBeans (EJB) bindings that are deployed in Service Component Architecture (SCA) applications in a cluster environment.

Service side

When an SCA service is exposed with a binding.ejb element, the service is exposed through an enterprise bean. During deployment, the SCA runtime generates a session bean for the service that is exposed through EJB binding. The caller of the composite service can invoke this service by accessing the generated EJB.

If the service is exposed through an EJB 2 bean, the EJB is bound at:

`ejb/sca/ejbbinding/`*`component_name`*`/`*`service_name`*

For example:

`ejb/sca/ejbbinding/CompanyComponent/Company`

If the service is exposed through an EJB 3.x bean, the EJB is bound at:

`ejb/sca/ejbbinding/`*`component_name`*`/`*`service_name#fullyQualifiedServiceInterfaceName`*`Remote`

For example:

`ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.CompanyRemote`

The generated EJB for the composite service will be under *profile_root*`/installedApps/`*`cell_name`*`/`*`composite_name`*`.ear/`.

Callers need to include client required classes (such as remote or home) of this generated bean in their classpath (or bundle the classes in their JAR file).

Lookup and invoke of this generated service EJB from a clustered environment is the same as lookup and invoke of any EJB in a product clustered setup. Refer to "Naming considerations in clustered and cross-server environments" in the *EJB 3.x application bindings overview* topic.

Reference side

When used on the reference side, the binding.ejb element should specify a URI attribute with values that match the value that is typically used when an EJB client calls the initialContext.lookup() method. The general convention is:

`"corbaname:iiop:`*`host`*`:`*`port`*`/NameServiceServerRoot#`*`JNDI_name`*`"`

where *JNDI_name* is the JNDI name of the target EJB.

For example:

`uri="corbaname:iiop:localhost:2809/NameServiceServerRoot#ejb/session/PriceQuoteSessionFacadeBean"`

JNDI name syntax differs if the target EJB is an EJB 2 or EJB 3.x bean.

When the referred EJB service is in a different cell, the URI might resemble one of the following:

`uri="corbaname:iiop:localhost:2809/NameServiceServerRoot#cell/clusters/cluster1/ejb/session/PriceQuoteSessionFacadeBean"`

or

`uri="corbaname::/NameServiceServerRoot#cell/clusters/cluster1/ejb/session/PriceQuoteSessionFacadeBean"`

or

`uri="cell/clusters/cluster1/ejb/session/PriceQuoteSessionFacadeBean"`

if the target EJB is on the same machine but on different cluster.

In advanced scenarios on multiple-server environments, a simpler and more portable way to access the target EJB application from an SCA composite is to set up a namespace binding and use the namespace binding name in the URI attribute of the binding.ejb along with `cell/persistent/`. For example:

```
uri="cell/persistent/PriceQuote"
```

where PriceQuote is the `name` field in the namespace.

The namespace binding can be of type EJB or CORBA based on the advanced scenario.

If the target EJB application which the composite is trying to access is on same cell, but on a different server, node or cluster, configure an EJB namespace binding. You can do this from the administrative console:

1. Click **Environment** > **Naming** > **Name space bindings**.
2. Select the cell scope.
3. Click **New**.
4. On the Specify binding type page, select the **EJB** binding type.
5. On the Specify basic properties page, specify the binding identifier, name in namespace, enterprise bean location such as server cluster or single server (with node), and JNDI name as needed. Use the **Name in name space** field to construct the URI as `cell/persistent/`*`name_in_namespace`*.

If the composite is running on a Version 7.0 cell and the target EJB application is running on a Version 6.1 product, configure a CORBA namespace binding with the correct Corbaname URL of the target EJB. Example Corbaname URL syntax is:

```
"corbaname:iiop:host:port/NameServiceServerRoot#jndi_name"
```

After you configure the namespace binding, use the **Name in name space** field to construct the URI; for example, `uri="cell/persistent/PriceQuote"` where PriceQuote is the value in the **Name in name space** field of the binding.

An advantage of using a namespace binding is, even when the target EJB changes, the composite definition does not need to change. Only the namespace binding needs to change accordingly.

SCA EJB reference binding URI patterns

Use an EJB reference binding URI pattern that is appropriate for the user setup and configuration:

- If an EJB reference binding accesses a stateless session bean on the same server, the EJB reference binding URI can be a Java Naming and Directory Interface (JNDI) name of the stateless session bean. For example:

  ```
  uri="ejb/com/app/resumebank/ResumeBankHome"
  ```

- If the EJB reference binding references another SCA service with the EJB binding on the same server, then the EJB reference binding URI can be either of the following:
  - The JNDI name of the stateless session bean. For example,

    ```
    uri="ejb/com/app/resumebank/ResumeBankHome"
    ```

  - A binding that uses the SCA reference target mechanism instead of the binding URI:

    ```
    <reference name="service1" target="component_name/service_name">
    ....
    <binding.ejb/>
    </reference>
    ```

- If the EJB reference binding is accessing a stateless session bean that is deployed in the same cell, the EJB reference binding URI can be based on the cluster/node/server setup. See the following examples:
  - Accessing a stateless session bean deployed in the same cell and on cluster2

    ```
    uri="cell/clusters/cluster2/ejb/com/app/resumebank/ResumeBankHome"
    ```

  - Accessing a stateless session bean deployed in the same cell but on a different node.

```
uri="cell/nodes/node_name/servers/server_name/ejb/com/app/resumebank/ResumeBankHome"
```

- If an EJB reference binding accesses a stateless session bean on a cell other than the current cell or on a mixed cell, then you can configure a reference binding URI in either of the following ways:
  - Create a namespace binding of either a Corba or EJB type using the administrative console. After configuring the namespace binding, use the **Name in name space** value to construct the URI as:

    ```
    uri="cell/persistent/name_in_namespace_binding"
    ```

    For example, if `neworder` is the name in the namespace binding, then the URI is `uri="cell/persistent/neworder"`.
  - Use corbaname for the EJB reference binding URI. For example:

    ```
    corbaname:iiop:<hostName><port>/NameServiceServerRoot#jndi_name
    ```

    where *hostName* and *port* are the respective host and port, and where the target stateless session bean is running. *jndi_name* is the JNDI name of the EJB which differs between EJB2 and EJB3.

Configuring EJB bindings in SCA OASIS applications

You can configure Enterprise JavaBeans (EJB) service and reference bindings for use in SCA OASIS applications.

About this task

Note: The SCA stateless session bean binding is a protocol binding that you can use to integrate SCA with EJB-based services. The product provides OASIS support for both EJB 3.x and 2.x bindings for services and references.

- Configure an EJB service binding to expose an SCA service as a stateless session bean for consumption by Java Platform, Enterprise Edition (Java EE) applications.
- Configure an EJB reference binding to enable SCA components to invoke stateless session beans.

Procedure

- Configure EJB service bindings and invoke them in caller applications.

  Steps follow for configuring EJB 3.x and 2.x bindings:

  EJB 3.x binding

  1. Configure an EJB 3.x binding in a composite definition.

     The following example shows an OASIS composite definition that has a service exposed over an EJB 3.x binding:

     ```
     <composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
                targetNamespace="http://www.ibm.com/samples/sca/ejb/jobbank"
                name="JobBank">
        <component name="CompanyComponent">
           <implementation.java class="com.app.jobbank.CompanyImpl"/>
           <!-- Expose service over the EJB 3.x binding -->
              <service name="Company">
                 <interface.java interface="com.app.jobbank.Company"/>
                 <binding.ejb/>
              </service>
        </component>
     </composite>
     ```

     The default ejb-version is `EJB3` in an SCA OASIS implementation. You do not need to specify ejb-version. However, you can explicitly define ejb-version; for example:

     ```
     <binding.ejb ejb-version="EJB3"/>
     ```

     Do not specify the uniform resource identifier (URI) for the EJB service binding in the composite definition. The run time will generate a unique URI, which is used as the JNDI name for the session bean endpoint that is generated for the SCA service, and ignore any URI for the EJB service binding in the composite definition.

2. Package the SCA composite with the service interface and implementation in a Java archive (JAR) and deploy the JAR in a business-level application.

   During SCA service startup, the run time generates a stateless session bean endpoint in memory that a Java EE client can invoke using the Java EE client programming model.

   The JNDI name for a generated session bean has the following format:

   ejb/sca/ejbbinding/*component_name*/*service_name*#*package.qualified.sca.interface*

   The JNDI name for the generated session bean for the jobbank composite service in step 1 is:

   ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.Company

3. Invoke the SCA service from a Java EE client application.

   The following code shows how to lookup and invoke an SCA service exposed as an EJB 3.x binding from a Java EE caller application:

```
// Look up EJB for the SCA service
Object remoteObj =
   initialContext.lookup("ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.Company");
Company company = (Company) PortableRemoteObject.narrow(remoteObj, Company.class);
// Invoke component service by invoking EJB method
String result = company.getCompanyInfo("ACME");
```

   The Java EE caller application should use an SCA service interface as an EJB remote interface for the lookup and invocation. In the sample code, `Company` is an SCA service or business interface that is declared under the `<interface.java>` element. Thus, the only class required by the Java EE caller application is an SCA service or business interface.

   The SCA service can be looked up based on the WebSphere topology like a typical EJB; for example:

"cell/nodes/*node_name*/servers/*server_name*/ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.Company"

"cell/clusters/*cluster_name*/ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.Company"

"corbaname:iiop:host:port/NameServiceServerRoot#ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.Company"

EJB 2.x binding

1. Configure an EJB 2.x binding in a composite definition.

   The following example shows an OASIS composite definition that has a service exposed over an EJB 2.x binding:

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
           targetNamespace="http://www.ibm.com/samples/sca/ejb/jobbank"
           name="JobBank">
   <component name="CompanyComponent">
      <implementation.java class="com.app.jobbank.CompanyImpl"/>
      <!-- Expose service over the EJB 3.x binding -->
      <service name="Company">
         <interface.java interface="com.app.jobbank.Company"/>
         <binding.ejb ejb-version="EJB2"/>
      </service>
   </component>
</composite>
```

   You must explicitly define ejb-version as `EJB2`.

   Do not specify the URI for the EJB service binding in the composite definition. The run time will generate a unique URI, which is used as the JNDI name for the session bean endpoint that is generated for the SCA service, and ignore any URI for the EJB service binding in the composite definition.

2. Package the SCA composite with the stateless session bean service binding in a JAR and deploy the JAR in a business-level application.

   During deployment, the SCA run time generates a 2.x stateless session bean. The JNDI name pattern for a generated session bean is:

   ejb/sca/ejbbinding/*component_name*/*service_name*

   The JNDI name for the generated session bean for the jobbank composite service in step 1 is:

   ejb/sca/ejbbinding/CompanyComponent/Company

The generated session bean for the composite service is under the directory, *profile_root*/installedApps/*cell_name*/*sca.composite.name*App.ear; for example: *profile_root*/installedApps/*cell_name*/JobBankApp.ear.

3. Invoke the SCA service from a Java EE client application.

   The Java EE caller application invoking this SCA service should do either of the following:

   – Bundle generated classes from the directory in step 2 with the client application.

   – Add the generated EJB module JAR file from the directory in step 2 to the client class path.

   If the caller application is running on a non-WebSphere application server, a "Bare" Java Standard Edition (SE), or a version of WebSphere Application Server previous to version 7.0 without the Feature Pack for EJB 3.0, then you must run the **createEJBStubs** command for the generated EJB module to generate client-side stubs and include the generated subs at the client application. The **createEJBStubs** command is under the *profile_root*/installedApps/*cell_name*/*sca.composite.nameApp.ear*/ directory. See the topic on the create stubs command.

   If the caller of the SCA service is another SCA reference with an EJB binding, the caller SCA composite does not need to include the generated home, remote interface because the SCA reference binding implementation can invoke the SCA service binding implementation using only the SCA interface.

- Configure EJB reference bindings.

  An EJB reference binding enables SCA components to invoke stateless session beans.

  The following example shows an SCA composite definition that contains an SCA reference with an EJB binding:

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
           targetNamespace="http://www.ibm.com/samples/sca/ejb/jobbank"
           name="JobBank">

  <component name="CompanyComponent">

     <reference name="extEJBService">
        <interface.java interface="com.app.resumebank.ResumeBank"/>
        <binding.ejb uri="ejb/com/app/resumebank/ResumeBankHome"/>
     </reference>

  </component>
</composite>
```

  An EJB reference binding supports invocation of both local and remote stateless session beans.

  The binding URI should be the JNDI name of the stateless session bean which the SCA reference is trying to invoke. The JNDI name of the session bean can be in the EJB 2.x or EJB 3.x format based on whether the target EJB version is 2.x or 3.x. The JNDI name can be local if the target is a local session bean or remote if the target is remote.

  You do not need to specify an ejb-version attribute under the <binding.ejb> element for an EJB reference binding.

  Also, you do not need to package the target session bean home interface or client stubs with an SCA component with an EJB reference binding. The EJB session bean binding implementation can dynamically look up, create, and invoke the bean without the usual EJB client classes.

  For the SCA reference interface, use either the target EJB interface or its compatible interface. Interface compatibility rules are defined under Section 3.1, *Compatibility of Interfaces used for SCA Services & References with EJB Session Bean Interfaces*, of the EJB Session Bean Binding Specification 1.1.

  Some of the supported binding `uri` formats are in the following list. For binding `uri`, follow EJB 2.x or EJB 3.x JNDI lookup name formats based on the target EJB type.

  Corba url format

```
<binding.ejb uri="corbaname:iiop:host:port/NameServiceServerRoot#jndi_name"/>
```

  Local session bean

```
<binding.ejb uri="ejblocal:jndi_name"/>
```

  uri based on the topology

Cluster

```
<binding.ejb uri="cell/clusters/cluster_name/jndi_name" />
```

Server

```
<binding.ejb uri="cell/nodes/node_name/servers/server_name/jndi_name" />
```

Persistent namespace

```
<binding.ejb uri="cell/persistent/name_in_persistent_namespace_binding"/>
```

Results

You have configured an EJB service or reference binding for an SCA OASIS application.

Use of the bindings has the following limitations:

- A local EJB invocation from a Java EE application to an SCA service with an EJB 2.x or EJB 3.x binding is not supported.
- A Java EE application running on a non-WebSphere application server, on “bare” Java Standard Edition (SE), or on a WebSphere application server previous to Version 7.0 without the Feature Pack for EJB 3.0 cannot invoke the SCA EJB 3.x service binding.
- The product does not support an SCA composite that has both EJB 2.x and EJB 3.x services in different components but implements the same interface.

What to do next

Ensure that your components do not use an SCA OASIS implementation that the product does not support.

Table 87. Unsupported sections of SCA OASIS specifications. The product does not support these sections of the SCA EJB Session Bean Binding specification.

Section	Not supported in SCA OASIS implementation

2 Session Bean Binding Schema	`binding.ejb/@uri` for SCA service binding
	• The SCA stateless EJB implementation automatically generates a unique URI based on the component name, service name, and service interface. The URI is used as the EJB service endpoint.
	See step 1 in the "Configure EJB service bindings and invoke them in caller applications" procedure.
	`corbaname:rir:#ejb/MyHome`
	• The product supports the `cobra url` format:
	`corbaname:iiop:host:port/NameServiceServerRoot#jndi_name`
	`ejb-link-name`
	• The product does not support `ejb-link-name` in the SCA OASIS implementation.

Configuring the SCA JMS binding

You can configure the Service Component Architecture (SCA) Java Message Service (JMS) binding for services and references to support messaging between SCA applications and JMS providers.

Before you begin

Use the administrative console to enable the SIB service and **Configuration reload enabled** option on the application server where the application runs. Restart the server to enable the dynamic reloading of the SIB configuration files for this server. See the SIB service settings information to learn more about enabling the SIB service.

When you use the JMS binding, it is important that you follow best practice guidelines on design, configuration, and tuning of the messaging topology. Following these guidelines is especially important when you design systems for high throughput or high availability. Read the documentation on using JMS bindings in this information center, specifically “Multiple-server bus with clustering.” The developerWorks articles “Configuring and tuning WebSphere MQ for performance on Windows and UNIX” and “Performance tuning for Java Messaging Service on WebSphere Application Server on z/OS” also provide useful information.

About this task

Bindings determine how a component communicates with the world outside its domain. SCA services use bindings to describe the access mechanism that clients must use to call the service. SCA references use bindings to describe the access mechanism used to call a service.

Using the SCA JMS binding, you can make SCA components available over JMS or you can use existing JMS applications within an SCA environment. You can use the SCA JMS binding element, `<binding.jms>`, within either a component service or a component reference definition. When a JMS binding is applied within a component service interface definition, the JMS binding enables clients to access an SCA service that is offered by a JMS provider. When the JMS binding is applied on a component reference, the SCA component can consume an external JMS application or another SCA component using JMS.

WebSphere Application Server supports asynchronous messaging using JMS. The default messaging provider enables enterprise applications deployed on WebSphere Application Server to perform asynchronous messaging without the need for you to install a JMS provider. The default messaging provider is installed and runs as part of WebSphere Application Server. The product supports the default messaging provider or WebSphere MQ as the messaging engine.

The SCA JMS Binding specification describes the `<binding.jms>` binding element and available attributes and options. To learn more about the `<binding.jms>` binding element, see the SCA JMS Binding specification documentation.

The product supports both the OASIS and the OSOA SCA JMS binding specification. Unless otherwise specified, the information in this topic pertains to applications for both the OSOA and OASIS specification.

SCA with JMS supports the following messaging exchange patterns:

- request-response messaging
- one-way messaging
- one-way messaging with callback

A *request* is a message that is sent to an SCA service or sent by an SCA reference. A *response* is a message that is received back at a reference or a message that is sent by a service in response to a previous request message. In SCA, a response is always a reply to a previous request. In a one-way message, a request message is sent and a response is not expected.

This task describes the steps necessary to enable SCA applications for JMS using request-response or one-way messaging.

Procedure

1. Identify the SCA business-level application that you want to enable for JMS messaging.
2. Identify and configure the JMS resources for your SCA application.

   You can configure JMS resources for the default messaging provider before deployment of the SCA application. During deployment of the SCA application, the product can dynamically create any JMS resources that the JMS binding needs that do not exist if the composite definition file is configured for dynamic resource creation.

SCA applications are bound to JMS resources through their binding definitions in a corresponding composite definition file. The composite definition file for JMS uses Java Naming and Directory Interface (JNDI) names to identify JMS administered objects that are used by the SCA runtime environment on behalf of the SCA application to provide access over the specified binding.

You can configure the following JMS resource references in the JNDI directory or choose for the product to dynamically create the resources for you:

- a service integration bus
- a request queue and a response queue. A response queue is only required for request-response messaging. The physical queues and the logical queues must be defined.
- an activation specification to handle the request to the service
- a connection factory to process the response

To learn more about manually creating JMS resources, see the documentation on JMS resources for the default messaging provider. To learn more about dynamically creating JMS resources, see the dynamic JMS resource creation during deployment information.

3. Configure an SCA service with the SCA JMS binding.

   To expose an SCA service over the JMS binding, add the `<binding.jms>` element within the service definition.

   Within the `<binding.jms>` element, define the following:

   - an `<activationSpec>` element to identify the activation specification that the binding uses to connect to a JMS destination to process request messages
   - a response `<connectionFactory>` element for request-response messaging patterns
   - a response `<destination>` element that describes the JMS destination for responses from the JMS binding. Specifying the response destination is optional because the runtime environment obtains the JMSReplyTo destination that is set on the incoming request message to determine the response destination. If the JMSReplyTo attribute is not set on the request message, the response `<destination>` element is used to identify the response destination. This element is not applicable for one-way messaging patterns.

4. Configure an SCA reference with the SCA JMS binding.

   To consume or expose a JMS application or SCA reference with the JMS binding, add the `<binding.jms>` element within the SCA reference definition.

   Within the `<binding.jms>` element, define the following:

   - a `<connectionFactory>` element to identify the JNDI name of the connection factory used to process messages sent from the reference to the referenced service
   - a `<destination>` element to identify the JMS queue or topic that is used to send messages to the referenced service
   - a response `<destination>` element that describes the JMS destination queue used to process responses from the JMS binding. This element is optional for request-response messaging and not applicable for one-way messaging

   Note: Differences between the OSOA and OASIS JMS binding specifications might affect migration of applications from OSOA to OASIS. Some commonly encountered differences include:

   - The validation schemas used for the OASIS applications are defined by the OASIS SCA specification. The OASIS binding schema definition mandates that binding elements appear in the exact order as defined in the schema. For OSOA applications, element order does not affect validation so, when migrating applications from OSOA, ensure the binding elements are in the order specified by the OASIS binding schema.
   - For the `<destination>`, `<connectionFactory>`, and `<activationSpec>` elements, the name attribute is `jndiName` in OASIS.
   - In OASIS composites, callback references must specify a destination element. The specified destination is used only if the destination cannot be determined from the

scaCallbackDestination or JMSReplyTo properties of the service request message. In OSOA composites, callback references can omit the destination element if it is known that one of the properties is always set.

For information on other differences, consult the OASIS specification for additional information about specific elements.

5. Optional: Configure JMS message correlation.

   You can optionally configure a JMS correlation identification for a request message and its response. Because of the asynchronous nature of JMS, it is important to determine if message correlation is required. A request-response messaging pattern is not equivalent to synchronous messaging.

   You can use the @correlationScheme element to identify the correlation scheme used when sending a reply or callback messages.

   For OSOA, the valid values for this element are: `RequestMsgIDToCorrelID`, `RequestCorrelIDToCorrelID`, and `none`. The default value is `RequestMsgIDToCorrelID`.

   For OASIS, the valid values for this element are: `sca:messageID`, `sca:correlationID`, and `sca:none`.

   To configure JMS message correlation, you can specify a `correlationID` element within the composite definition file that is used within the JMS header to link two messages.

6. Optional: Invoke an operation using JMS operation selection
   - Use the default JMS binding operation selection.
   - Use JMS user property operation selection.
   - Use a JMS custom operation selector.

   Using the @nativeOperation attribute in a composite definition, your application can override operation selection; for example, in OSOA:

```
<service>
  <binding.jms>
  ...
    <operationProperties
      name="operationToInvoke"
      nativeOperation="selectedOperation"/>
  ...
  </binding.jms>
</service>
```

   Or, in OASIS:

```
<service>
  <binding.jms>
  ...
    <operationProperties
      name="operationToInvoke"
      selectedOperation="selectedOperation"/>
  ...
  </binding.jms>
</service>
```

   In this example, when the configured operation selector selects the operation named *selectedOperation*, the run time invokes operation *operationToInvoke* on the target service implementation. The overriding occurs whether the service binding is configured using default JMS operation selection, JMS user property operation selection, or a JMS custom selector.

Results

You have defined the JMS resources you need for your SCA application, configured the SCA JMS binding for your SCA services and references, and deployed your application. Your application is now ready for use.

Example

The following example illustrates two SCA component implementations and the use of request-response and one-way messaging.

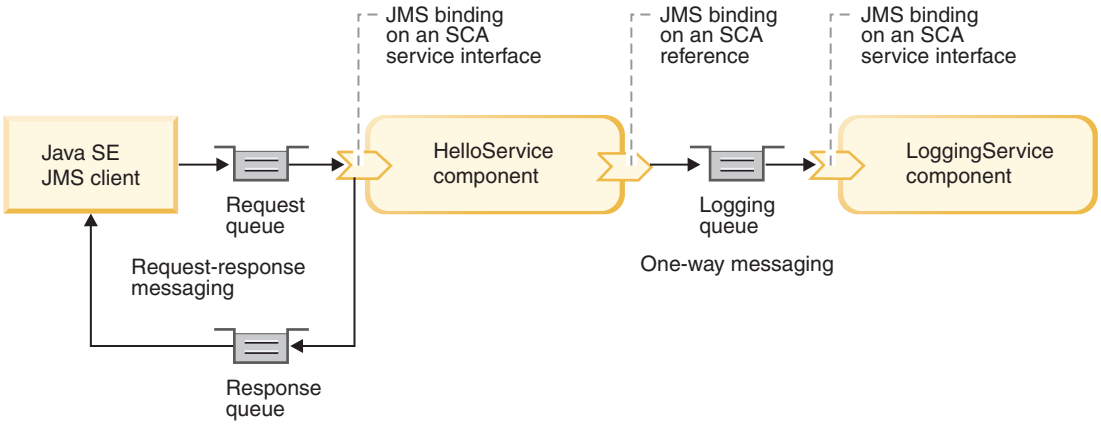


Figure 4. SCA JMS binding example

The HelloService component implementation illustrates the request-response message pattern. This HelloService component exposes the service interface with the name, `getGreeting`, that is used to illustrate a return response of `hello` plus the value of `getGreeting`.

The LoggingService component implementation is a logging service. This component exposes a one-way service interface with the name, `log`, that receives a message and logs the message in a repository.

The HelloService has an SCA reference to the LoggingService. Each time the HelloService service receives a message, it calls the LoggingService service to log the message.

In this example, a Thin Client for JMS application sends a message, formatted as a JMS `ObjectMessage` message type to the SCA HelloService using the `jms/SCA_sample_Request` queue. The ObjectMessage sets the `scaOperationName` property to the value, `getGreetings`. The HelloServiceComponent receives the message over the JMS HelloService binding. The HelloServiceComponent then sends a request to the referenced service, LoggingService, and the one-way operation is complete. HelloServiceComponent sends a response of `hello` plus the value of `getGreetings` to the client application using the `jms/SCA_sample_Response` queue to complete the request-response operation.

Configuring an SCA JMS binding for a request-response service from a JMS client to SCA service

The following example describes a `<binding.jms>` element within the component definition file for a request-response message exchange pattern from a JMS client to an SCA service:

OSOA composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="http://www.ibm.com/soa/sca/samples"
   xmlns:hw="http://www.ibm.com/soa/sca/samples"
  xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.0"
   name="HelloServiceComposite">

  <component name="HelloServiceComponent">
    <implementation.java class="soa.sca.samples.jms.HelloServiceImpl"/>
    <service name="HelloService">
       <interface.java interface="soa.sca.samples.jms.HelloService"/>
         <binding.jms>
           <ts:wireFormat.jmsObject/>
           <destination name="jms/SCA_sample_Request" type="queue"/>
           <activationSpec name="jms/SCA_sample_AS"/>
           <response>
```

```
            <destination name="jms/SCA_sample_Response" type="queue"/>
            <connectionFactory name="jms/SCA_sample_CF"/>
          </response>
        </binding.jms>
    </service>
  </component>

</composite>
```

OASIS composite

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
  targetNamespace="http://www.ibm.com/soa/sca/samples"
   xmlns:hw="http://www.ibm.com/soa/sca/samples"
  xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.1"
   name="HelloServiceComposite">

  <component name="HelloServiceComponent">
    <implementation.java class="soa.sca.samples.jms.HelloServiceImpl"/>
    <service name="HelloService">
       <interface.java interface="soa.sca.samples.jms.HelloService"/>
         <binding.jms>
           <ts:wireFormat.jmsObject/>
           <destination jndiName="jms/SCA_sample_Request" type="queue"/>
           <activationSpec jndiName="jms/SCA_sample_AS"/>
           <response>
              <destination jndiName="jms/SCA_sample_Response" type="queue"/>
              <connectionFactory jndiName="jms/SCA_sample_CF"/>
           </response>
        </binding.jms>
    </service>
  </component>

</composite>
```

- The <destination> describes the JMS destination. The destination type is either a queue or topic. This example illustrates the JMS destination queue type. The destination is used to process requests by the JMS binding to the component implementation that contains the service interface
- The <activationSpec> element identifies the activation specification that the binding uses to connect to a JMS destination to process request messages. The activation specification name must be a JNDI name. The <activationSpec> element is only supported within the SCA <service> tag.
- The <response> element defines the resources used for processing response messages. In this example, the response element specifies the resources for sending messages from the <service> back to the client.
- The response <destination> element describes the JMS destination queue that is used to process responses from the service interface.
- The response <connectionFactory> element identifies the JNDI name of the connection factory that the binding uses to process response messages.
- The <ts:wireFormat.jmsObject/> specifies to the JMS binding that the payload of the message is of the type, javax.jms.Message (ObjectMessage).

The following example describes a <binding.jms> element within the component definition file for a one-way interaction from one SCA service to another SCA service. The <binding.jms> binding definition is similar to the HelloService; however, because the message operation is one-way, there is not a response definition.

OSOA composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="http://www.ibm.com/soa/sca/samples"
  xmlns:hw="http://www.ibm.com/soa/sca/samples"
  xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.0"
  name="HelloServiceComposite">

 <component name="LoggingService">
```

```
  <implementation.java class="soa.sca.samples.jms.LoggingServiceImpl"/>

  <service name="LoggingService">
   <interface.java interface="soa.sca.samples.jms.LoggingService"/>
         <binding.jms>
           <ts:wireFormat.jmsObject/>
           <destination name="jms/SCA_Logging_Request" type="queue"/>
           <activationSpec name="jms/SCA_sample_AS"/>
         </binding.jms>
  </service>

 </component>
</composite>
```

OASIS composite

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
  targetNamespace="http://www.ibm.com/soa/sca/samples"
  xmlns:hw="http://www.ibm.com/soa/sca/samples"
  xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.1"
  name="HelloServiceComposite">

 <component name="LoggingService">
  <implementation.java class="soa.sca.samples.jms.LoggingServiceImpl"/>

  <service jndiName="LoggingService">
   <interface.java interface="soa.sca.samples.jms.LoggingService"/>
         <binding.jms>
           <ts:wireFormat.jmsObject/>
           <destination jndiName="jms/SCA_Logging_Request" type="queue"/>
           <activationSpec jndiName="jms/SCA_sample_AS"/>
         </binding.jms>
  </service>

 </component>
</composite>
```

Configuring an SCA JMS binding for two-way service from a JMS client to SCA reference

The following example describes a <binding.jms> element within the component definition file for a request-response message exchange pattern from a JMS client to an SCA reference:

OSOA SCA reference

```
<reference name="helloWorldService">
    <interface.java interface="my.HelloWorldService"/>
    <binding.jms>
       <connectionFactory name="jms/helloWorldServiceCF"/>
       <destination name="jms/HelloWorldService"/>
       <response>
           <destination name="jms/SCA_sample_Response"/>
       </response>
    </binding.jms>
</reference>
```

OASIS SCA reference

```
<reference name="helloWorldService">
    <interface.java interface="my.HelloWorldService"/>
    <binding.jms>
        <connectionFactory jndiName="jms/helloWorldServiceCF"/>
        <destination jndiName="jms/HelloWorldService"/>
        <response>
            <destination jndiName="jms/SCA_sample_Response"/>
        </response>
    </binding.jms>
</reference>
```

- The <connectionFactory> element is used in the SCA reference to identify the JNDI name of the connection factory used to process messages sent from the reference to the referenced service. The <activationSpec> element is not supported in a reference.
- The <destination> element is the JMS queue or topic that is used to send messages to the referenced component implementation.
- The response <destination> element is the JMS resource that is used to receive response messages to the SCA reference.

Authenticating with a secure bus

You can specify an authentication alias on the reference binding or service binding in the composite file to authenticate with a secure bus. Use the authentication-alias attribute to specify the predefined authentication alias instead of specifying it in the activation specification or connection factory settings.

A <binding.jms> element does not propagate the identity of the client. For details on creating a J2C authentication alias, see Managing Java 2 Connector Architecture authentication data entries for JAAS.

OSOA SCA composite

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://www.ibm.com/soa/sca/samples"
    xmlns:hw="http://www.ibm.com/soa/sca/samples"
    xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.0"
    xmlns:websphere="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
    name="HelloServiceComposite">
  <component name="HelloServiceComponent">
    <implementation.java class="soa.sca.samples.jms.HelloServiceImpl"/>
    <service name="HelloService">
      <interface.java interface="soa.sca.samples.jms.HelloService"/>
      <binding.jms websphere:authentication-alias="SCA_Auth_Alias">
        <destination name="jms/SCA_sample_Request" type="queue"/>
        <activationSpec name="jms/SCA_sample_AS"/>
        <response>
          <destination name="jms/SCA_sample_Response" type="queue"/>
          <connectionFactory name="jms/SCA_sample_CF"/>
        </response>
        <ts:wireFormat.jmsObject/>
      </binding.jms>
    </service>
  </component>
</composite>
```

OASIS SCA composite

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
    targetNamespace="http://www.ibm.com/soa/sca/samples"
    xmlns:hw="http://www.ibm.com/soa/sca/samples"
    xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.1"
    xmlns:websphere="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"
    name="HelloServiceComposite">
  <component name="HelloServiceComponent">
    <implementation.java class="soa.sca.samples.jms.HelloServiceImpl"/>
    <service name="HelloService">
      <interface.java interface="soa.sca.samples.jms.HelloService"/>
      <binding.jms websphere:authentication-alias="SCA_Auth_Alias">
        <destination jndiName="jms/SCA_sample_Request" type="queue"/>
        <activationSpec jndiName="jms/SCA_sample_AS"/>
        <response>
          <destination jndiName="jms/SCA_sample_Response" type="queue"/>
          <connectionFactory jndiName="jms/SCA_sample_CF"/>
        </response>
        <ts:wireFormat.jmsObject/>
      </binding.jms>
    </service>
  </component>
</composite>
```

What to do next

Based on your business needs, you can configure an SCA JMS binding wire format or configure transactions for the SCA JMS binding.

Explore the JMS binding samples to better understand how to invoke an SCA component service using a JMS client. For information on JMS samples, see “SCA samples.” To download JMS binding samples from a product website:

1. Go to the **Samples, Version 8.5** information center.
2. On the **Downloads** tab, click **FTP** or **HTTP** in the **Service Component Architecture** section.
3. In the authentication window, click **OK**.
4. From the **SCA.zip** compressed file, download the `SCA/JMS` directory and follow instructions in `SCA/JMS/documentation/readme.html` to build the files.

   Instead of building deployable files, you can use the prebuilt `jms-callback-service.jar`, `jms-twoway-oneway-service.jar`, and `jms-twoway-service.jar` files in the `SCA/installableApps` directory of the compressed file.

After you obtain your SCA components, deploy the components in an application and test the JMS binding.

Configuring SCA JMS binding wire formats

You can configure the messaging data formats between an SCA application and a JMS producer or consumer, by configuring your SCA application to take advantage of a supported message type and data format.

Before you begin

Configure the JMS binding for your SCA application.

About this task

JMS producers and consumers use a variety of message types to hold application data payloads. You can use the JMS BytesMessage, ObjectMessage, or TextMessage message types. In some cases, the TextMessage message type might be expected to contain additional structure such as application data in serialized XML format.

Wire format describes the format of the data that is on the wire. For the SCA JMS binding, the wire format is the format of the data in the JMS message that flows through the JMS provider. Because of the variety of message types and formats, SCA services and references that are configured with a JMS binding might require additional configuration so that the runtime environment can perform the marshalling and unmarshalling required to translate between application data formats and the format of the JMS message on the wire. The additional configuration of message types is the specification of the wire format for message handling.

Whether you are configuring an SCA service or reference, it is important to recognize if the wire format is previously established by your existing messaging application infrastructure, or if you are selecting the wire format along with your SCA application. If you are starting with an application with a preexisting messaging infrastructure and you are adding your SCA application to this environment, the wire format is likely already determined by the messaging infrastructure. If you are starting with an SCA application and you intend for this application to interact with future JMS message producers or consumers, you can specify the wire format within your SCA application.

For the cases when the wire format is predetermined, it is expected behavior that the wire format enables a natural mapping to the format expected by your preexisting message producers and consumer when dealing with input and non-exception output. However, exception conditions might not be handled

according to the convention for the wire format type in your existing messaging application infrastructure. If you want exceptions to flow over the JMS binding, you might need to adjust the producers or consumers interacting with your SCA application over the JMS binding according to the exception handling behavior.

JMS binding wire format is supported for applications coded to both the OSOA and OASIS specifications. Unless otherwise specified, the information in this topic pertains to both OSOA and OASIS applications.

Procedure

1. Determine if you are using a wire format that is predetermined by your existing messaging infrastructure or if you are starting with an SCA application and defining the message wire format.
2. Determine the message type to use for your wire format.

   See Supported message types for SCA JMS binding wire formats.
3. If you are using a wire format predetermined by your existing messaging infrastructure, add the corresponding wire format element into the composition definition file.
4. Ensure that your SCA service and service client implementation and interfaces map appropriately for the specific wire format that you selected.
5. (optional) If you want exception checking to occur over the JMS binding, ensure that the JMS producer and consumer that is interoperating with your SCA application follows the SCA JMS binding exception handling procedures described previously.
6. If you are starting with an SCA application and defining the message infrastructure, add the appropriate wire format element into the composition definition file, and ensure that your future JMS producer or consumer applications understand how to interoperate with this message data format.

Results

You have configured the messaging data format between an SCA application and a JMS producer or consumer.

Examples of configuring SCA JMS binding wire formats

The following examples illustrate the configuration of composite definition files using the various wire format schemes. Component references are configured in an analogous manner

TextMessage with serialized XML wire format that maps data using JAXB

This example illustrates the TextMessage with serialized XML wire format that maps data using JAXB, which is the default wire format:

OSOA

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://test/soa/sca/"
    xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.0"
    name="JBC">

      <component name="JAXBComponent">
        <implementation.java class="test.backend.HelloWorldJAXBBackendImpl" />
        <service name="HelloWorldJAXBService">
          <interface.wsdl interface="http://test.hello#wsdl.interface(HelloWorld)"/>
          <binding.jms>
            <destination name="jms/Request1" type="queue" create="never"/>
            <activationSpec name="jms/AS1"/>
            <response>
              <destination name="jms/Response1" type="queue" create="never"/>
              <connectionFactory name="jms/CF" create="never"/>
            </response>
           <!-- No wire format element is necessary, the default is used. -->
```

```
        </binding.jms>
      </service>
    </component>
</composite>
```

OASIS

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
    targetNamespace="http://test/soa/sca/"
    xmlns:ts="http://tuscany.apache.org/xmlns/sca/1.1"
    name="JBC">

      <component name="JAXBComponent">
        <implementation.java class="test.backend.HelloWorldJAXBBackendImpl" />
        <service name="HelloWorldJAXBService">
          <interface.wsdl interface="http://test.hello#wsdl.interface(HelloWorld)"/>
          <binding.jms>
            <!-- No wire format element is necessary, the default is used. -->
            <destination jndiName="jms/Request1" type="queue" create="never"/>
            <activationSpec jndiName="jms/AS1"/>
            <response>
              <destination jndiName="jms/Response1" type="queue" create="never"/>
              <connectionFactory jndiName="jms/CF" create="never"/>
            </response>
         </binding.jms>
      </service>
    </component>
</composite>
```

ObjectMessage wire format that maps to the `java.lang.Object` class

This example illustrates the ObjectMessage wire format that maps to the `java.lang.Object` class:

OSOA

```
<component name="SerializableComponent">
    <implementation.java class="test.backend.HelloWorldSerializableBackendImpl"/>
    <service name="HelloWorldSerializableService">
        <interface.java interface="test.HelloWorldSerializableService"/>
        <binding.jms>
          <destination name="jms/Request2" type="queue" create="never"/>
          <activationSpec name="jms/AS2"/>
          <response>
            <destination name="jms/Response2"/>
            <connectionFactory name="jms/CF" create="never"/>
          </response>
          <!-- The wire format element must appear last in the binding definition
               to conform to the OSOA schema. -->
          <ts:wireFormat.jmsObject/>
        </binding.jms>
    </service>
</component>
```

OASIS

```
<component name="SerializableComponent">
    <implementation.java class="test.backend.HelloWorldSerializableBackendImpl"/>
    <service name="HelloWorldSerializableService">
        <interface.java interface="test.HelloWorldSerializableService"/>
        <binding.jms>
          <!-- The wire format element must appear first in the binding definition
               to conform to the OASIS schema. -->
          <ts:wireFormat.jmsObject/>
          <destination jndiName="jms/Request2" type="queue" create="never"/>
          <activationSpec jndiName="jms/AS2"/>
          <response>
            <destination jndiName="jms/Response2"/>
            <connectionFactory jndiName="jms/CF" create="never"/>
```

```
          </response>
        </binding.jms>
    </service>
</component>
```

TextMessage wire format that maps to single string messages

This example illustrates the TextMessage wire format that maps to single string messages:

OSOA

```
<component name="JMSTextBackendComponent">
    <implementation.java class="test.backend.HelloWorldTextBackendImpl" />
    <service name="HelloWorldTextService">
       <interface.java interface="test.HelloWorldTextService"/>
       <binding.jms>
          <destination name="jms/Request3"/>
          <activationSpec name="jms/AS3"/>
          <response>
             <destination name="jms/Response3"/>
             <connectionFactory name="jms/CF" create="never"/>
          </response>
          <!-- The wire format element must appear last in the binding definition
               to conform to the OSOA schema. -->
          <ts:wireFormat.jmsText/>
      </binding.jms>
    </service>
</component>
```

OASIS

```
<component name="JMSTextBackendComponent">
    <implementation.java class="test.backend.HelloWorldTextBackendImpl" />
    <service name="HelloWorldTextService">
       <interface.java interface="test.HelloWorldTextService"/>
       <binding.jms>
          <!-- The wire format element must appear first in the binding definition
               to conform to the OASIS schema. -->
          <ts:wireFormat.jmsText/>
          <destination jndiName="jms/Request3"/>
          <activationSpec jndiName="jms/AS3"/>
          <response>
             <destination jndiName="jms/Response3"/>
             <connectionFactory jndiName="jms/CF" create="never"/>
          </response>
      </binding.jms>
    </service>
</component>
```

BytesMessage wire format that maps to single byte array messages

This example illustrates the BytesMessage wire format that maps to single byte array messages:

OSOA

```
<component name="JMSBytesBackendComponent">
    <implementation.java class="test.backend.HelloWorldBytesBackendImpl" />
    <service name="HelloWorldBytesService">
      <interface.java interface="test.HelloWorldBytesService"/>
      <binding.jms>
        <destination name="jms/Request4"/>
        <activationSpec name="jms/AS4"/>
        <response>
          <destination name="jms/Response4"/>
          <connectionFactory name="jms/CF" create="never"/>
        </response>
        <!-- The wire format element must appear last in the binding definition
             to conform to the OSOA schema. -->
```

```
        <ts:wireFormat.jmsBytes/>
      </binding.jms>
    </service>
</component>
```

OASIS

```
<component name="JMSBytesBackendComponent">
    <implementation.java class="test.backend.HelloWorldBytesBackendImpl" />
    <service name="HelloWorldBytesService">
      <interface.java interface="test.HelloWorldBytesService"/>
      <binding.jms>
        <!-- The wire format element must appear first in the binding definition
             to conform to the OASIS schema. -->
        <ts:wireFormat.jmsBytes/>
        <destination jndiName="jms/Request4"/>
        <activationSpec jndiName="jms/AS4"/>
        <response>
          <destination jndiName="jms/Response4"/>
          <connectionFactory jndiName="jms/CF" create="never"/>
        </response>
      </binding.jms>
    </service>
</component>
```

JMS message wire format that does not extract the message payload and maps to the `javax.jms.Message` class

This example illustrates the JMS Message wire format that does not extract the message payload and maps to the `javax.jms.Message` class:

OSOA

```
<component name="JMSTextBackendComponent">
    <implementation.java class="test.backend.HelloWorldTextBackendImpl" />
    <service name="HelloWorldTextService">
      <interface.java interface="test.HelloWorldTextService"/>
      <binding.jms>
        <destination name="jms/Request4"/>
        <activationSpec name="jms/AS4"/>
        <response>
          <destination name="jms/Response4"/>
          <connectionFactory name="jms/CF" create="never"/>
        </response>
        <!-- The wire format element must appear last in the binding definition
             to conform to the OSOA schema. -->
        <ts:wireFormat.jmsText/>
      </binding.jms>
    </service>
</component>
```

OASIS

```
<component name="JMSTextBackendComponent">
    <implementation.java class="test.backend.HelloWorldTextBackendImpl" />
    <service name="HelloWorldTextService">
      <interface.java interface="test.HelloWorldTextService"/>
      <binding.jms>
        <!-- The wire format element must appear first in the binding definition
             to conform to the OASIS schema. -->
        <ts:wireFormat.jmsText/>
        <destination jndiName="jms/Request4"/>
        <activationSpec jndiName="jms/AS4"/>
        <response>
          <destination jndiName="jms/Response4"/>
          <connectionFactory jndiName="jms/CF" create="never"/>
        </response>
      </binding.jms>
    </service>
</component>
```

Examples of TextMessage messages using JAXB

The following examples illustrate the contents of TextMessage messages produced by using JAXB marshalling in the default wire format. These examples are based on using the bottom-up approach of developing SCA applications starting with a Java interface, rather than the best practice of starting with a WSDL interface, such that the runtime environment generates the WSDL file.

- For a client invoking an SCA reference with the following interface method:

```
package com.ibm.test.soa.sca;
public String getGreetings(String name);
```

  The binding produces a TextMessage request message with XML payload similar to the following:

```
<ns2:getGreetings xmlns:ns2="http://sca.soa.test.ibm.com/">
   <arg0>Mike</arg0>
</ns2:getGreetings>
```

  The response message must be a TextMessage with XML payload similar to the following:

```
<ns2:getGreetingsResponse xmlns:ns2="http://sca.soa.test.ibm.com/">
   <return>Hello Mike</return>
</ns2:getGreetingsResponse>
```

  Similarly, an SCA service bound to a destination, from which TextMessage messages of this format were delivered, requires the same service interface method; for example:

```
package com.ibm.test.soa.sca;
public String getGreetings(String name);
```

- For a client invoking an SCA reference with the following interface method:

```
package com.ibm.test.soa.sca;
public void getGreeting(FullName name, String string);
```

  In this example, the `FullName` type is mapped to Java using JAXB using the following schema definition:

```
<schema targetNamespace="http://www.ibm.com/test/soa/sca/"
    elementFormDefault="qualified"
    xmlns=http://www.w3.org/2001/XMLSchema>

    <complexType name="FullName">
      <sequence>
        <element name="firstName" type="string"/>
        <element name="lastName" type="string"/>
      </sequence>
    </complexType>
</schema>
```

  The binding produces a TextMessage request message with XML payload similar to the following:

```
<ns2:getGreetings xmlns:ns2="http://sca.soa.test.ibm.com/">
   <arg0>Mike</arg0>
</ns2:getGreetings>
```

  The response message must be a TextMessage request message with XML payload similar to the following:

```
<ns3:getGreeting xmlns:ns3="http://sca.soa.test.ibm.com/"
   xmlns:ns2="http://www.ibm.com/test/soa/sca/">
   <arg0>
        <ns2:firstName>Bob</ns2:firstName>
        <ns2:lastName>Jones</ns2:lastName>
   </arg0>
   <arg1>home</arg1>
</ns3:getGreeting>
```

  Similarly, an SCA service bound to a destination, from which TextMessage messages of this format were delivered, requires the same service interface method; for example:

```
package com.ibm.test.soa.sca;
public String getGreeting(FullName name, String string);
```

Supported message types for SCA JMS binding wire formats:

For a Service Component Architecture (SCA) application that has a Java Message Service (JMS) binding to send and receive messages, the application must use a message type and data format that the JMS binding supports. Supported message types include JMS BytesMessage, ObjectMessage, and TextMessage. The message type to use depends on whether you are configuring an SCA service or reference and whether the JMS wire format of the service or reference is already determined.

The supported message types apply to applications coded to both the OSOA and OASIS specifications. Unless otherwise specified, the information in this topic pertains to both OSOA and OASIS applications.

The following sections describe supported message types:

- Message types for SCA services when the JMS binding wire format is predetermined
- Message types for SCA references when the JMS binding wire format is predetermined
- Message types for SCA services and references when the JMS binding wire format is not predetermined

Message types for SCA services when the JMS binding wire format is predetermined

For an SCA service to consume messages that are produced in a predetermined JMS binding wire format, use the following message types:

- **TextMessage or BytesMessage with serialized XML wire format that maps data using JAXB**

  The TextMessage or BytesMessage wire format uses Java Architecture for XML Binding (JAXB) technology to marshal and unmarshal data into XML. This wire format is the default wire format. Thus, this wire format applies if no wire format element is specified in the composite definition file. To specify this wire format, add the following wire format element to the composite definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsdefault
  ```

  OASIS

  ```
  {http://docs.oasis-open.org/ns/opencsa/sca/200912}wireFormat.jmsDefault
  ```

  – **Mapping a request message payload to input arguments**

    The TextMessage or BytesMessage payload contains XML data that represents serialized input arguments. The format of the service is described by a WSDL interface that is specified by either a `interface.wsdl` element from the component service definition, or by a WSDL that is generated by the Java interface of the SCA service if the `interface.wsdl` element is not explicitly defined.

    This runtime environment Java-to-WSDL mapping is based on the JAXB and JAX-WS specifications. The default behavior results in a document-literal wrapped WSDL interface, except when there is a single parameter method. A single argument input parameter is **not** wrapped by a "wrapper" element corresponding to the operation name.

    JAXB unmarshalling is used to deserialize incoming XML payloads into the service arguments that are passed to the component implementation.

  – **Mapping an output return value to a response message payload**

    The return value of the service method is marshalled in a similar manner to the unmarshalling of the request message input. The marshalling uses JAXB and JAX-WS technology to map to XML in a WSDL defined format such that the WSDL is specified explicitly or the runtime environment generates the WSDL from the Java interface. The resulting serialized XML is never wrapped by the operation name, even if the WSDL defines a wrapper element.

    If the request payload is a BytesMessage, the non-wrapped XML is set as the payload of a response BytesMessage. If the request payload received is a TextMessage, the non-wrapped XML is set as a TextMessage. The response format is always the same as the request message format.

  – **Mapping exceptions to a response message payload**

    If you have a checked exception, the runtime environment will not marshall the exception. Instead, the runtime environment serializes the fault, as defined by JAX-WS, into an XML payload of a response TextMessage. If you have an unchecked exception, an instance of `RuntimeException` is set

as the payload of a response ObjectMessage. In both cases, a Boolean message property with the name, `org_apache_tuscany_sca_fault`, is set to`true` on the response message. To learn more about exceptions and faults, see the documentation on using business exceptions with SCA interfaces.

- **TextMessage with serialized XML wire format that maps data using JAXB**

  The TextMessage wire format uses JAXB technology to marshal and unmarshal data into XML. To specify this wire format, add the following wire format element to the composite definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsTextXML
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsTextXML
  ```

  – **Mapping a request message payload to input arguments**

    The TextMessage payload contains XML data that represents serialized input arguments. The format of the service is described by a Web Services Description Language (WSDL) interface that is specified by either a `interface.wsdl` element from the component service definition, or by a WSDL that is generated by the Java interface of the SCA service if the `interface.wsdl` element is not explicitly defined. This runtime environment Java-to-WSDL mapping is based on the JAXB and Java API for XML-Based Web Services (JAX-WS) specifications. This mapping results in a document-literal wrapped WSDL interface; however, the presence of the operation-level wrapper does not impact operation selection for the messages. JAXB unmarshalling is used to deserialize incoming XML payloads into the service arguments that are passed to the component implementation.

  – **Mapping an output return value to a response message payload**

    The return value of the service method is marshalled in a similar manner to the unmarshalling of the request message input. The marshalling uses JAXB and JAX-WS technology to map to XML in a WSDL defined format such that the WSDL is specified explicitly or the runtime environment generates the WSDL from the Java interface. The resulting serialized XML is the payload of a TextMessage response message.

  – **Mapping exceptions to a response message payload**

    If you have a checked exception, the runtime environment will not marshall the exception. Instead, the runtime environment serializes the fault, as defined by JAX-WS, into an XML payload of a response TextMessage. If you have an unchecked exception, an instance of `RuntimeException` is set as the payload of a response ObjectMessage. In both cases, a Boolean message property with the name, `org_apache_tuscany_sca_fault`, is set to`true` on the response message. To learn more about exceptions and faults, see the documentation on using business exceptions with SCA interfaces.

- **ObjectMessage wire format that maps to the `java.lang.Object` class**

  The ObjectMessage wire format uses serialized Java objects. To specify this wire format, add the following wire format element to the composition definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsObject
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsObject
  ```

  – **Mapping a request message payload to input arguments**

    The ObjectMessage payload returned by a `getObject()` method is typically an array that maps to the input parameters of your service method. In the case of a single non-array object, the object is mapped to the single argument of a method with only one parameter. The parameter type must be serializable by Java and implement the Java serialization interface, `java.io.Serializable`.

    For a method with exactly one parameter, the payload returned from `getObject()` can either be unwrapped, as described in the previous special case, or wrapped in an array of size one that maps to the single input parameter. When the runtime cannot determine if the payload array is a wrapper array that maps to the input parameters of your service method or if the payload array is the actual argument itself, the default behavior is to map the payload directly to the single argument of the

method. You can override this behavior by setting the `wrapSingle` attribute on the `wireFormat.jmsObject` element to `true`. When `wrapSingle` is set to `true`, the content of the payload array, which is a single element because the size of the array is one, is mapped to the single argument of the method. The payload array itself is not mapped to the method argument.

– **Mapping an output return value to a response message payload**

The return value of the service method is set as the ObjectMessage payload. The return type must also be Java-serializable and implement the Java serialization interface, `java.io.Serializable`. In the case of a void return type, the response payload is set to the value, `null`.

– **Mapping exceptions to a response message payload**

If you have a checked exception, the runtime environment sets the exception as the ObjectMessage payload. For an unchecked exception, an instance of `RuntimeException` is set as the payload of a response ObjectMessage. In both cases, a Boolean message property with name `org_apache_tuscany_sca_fault` is set to `true` on the response message.

- **TextMessage wire format that maps to single `String` messages**

Use the `wireFormat.jmsText` format to map between the payload of a TextMessage and a single `String` argument or return value. To specify this wire format, add the following wire format element to the composition definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsText
```

OASIS

```
{http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsText
```

– **Mapping a request message payload to input arguments**

When using this wire format, the service interface method must contain a single parameter of type `String`; for example: `void myMethod(String str)`. The TextMessage payload that is returned by the `getText()` method is mapped to this single parameter.

– **Mapping an output return value to a response message payload**

The return type of the service method is typically either `String` or `void`. If the return type is `String`, the return value is set as the response TextMessage payload. If the return type is `void`, a `null` value is set as the payload. For a return value of type other than `String`, the payload is defined by setting a `String.valueOf()` on the return value.

– **Mapping exceptions to a response message payload**

Rather than defining a convention for mapping exceptions to a TextMessage, ObjectMessage messages are used to map exceptions. If you have a checked exception, the runtime environment sets the exception as the ObjectMessage payload. For an unchecked exception, an instance of `RuntimeException` is set as the payload of a response ObjectMessage message. In both cases, a Boolean message property with name `org_apache_tuscany_sca_fault` is set to `true` on the response message.

- **BytesMessage with XML wire format that maps data using JAXB**

The BytesMessage wire format uses JAXB technology to marshal and unmarshal data into XML. To specify this wire format, add the following wire format element to the composite definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsBytesXML
```

OASIS

```
{http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsBytesXML
```

– **Mapping a request message payload to input arguments**

The input arguments are marshalled into an XML payload, and added to a request BytesMessage. The format of the service is described by a WSDL interface that is specified by either a `interface.wsdl` element from the component service definition, or by a WSDL that is generated by the Java interface of the SCA service if the `interface.wsdl` element is not explicitly defined. This runtime environment Java-to-WSDL mapping is based on the JAXB and JAX-WS specifications. This mapping results in a document-literal wrapped WSDL interface; however, the presence of the

operation-level wrapper does not impact operation selection for the messages. The product uses JAXB marshalling to serialize the Java client input argument into the XML payload.

– **Mapping exceptions to a response message payload**

The runtime environment first determines whether the response message represents an unchecked exception condition by looking for the `org_apache_tuscany_sca_fault` Boolean message property. If this property is set to `true` on the response message, the runtime environment expects an ObjectMessage as the response. When this condition occurs, a ServiceRuntimeException error is returned to the client. If this property is not set, or set to `false`, the message is a BytesMessage that can represent either a checked exception or normal output data. If a checked exception, the BytesMessage payload contains a fault in XML serialized form. The runtime environment deserializes the fault using JAXB unmarshalling, and returns the corresponding checked exception back to the client, wrappering the fault data. For more information about exceptions and faults, see the topic on using business exceptions with SCA interfaces.

– **Mapping an output return value to a response message payload**

The return value of the service method is marshalled in a similar manner to the unmarshalling of the request message input. The marshalling uses JAXB and JAX-WS technology to map to XML in a WSDL defined format such that the WSDL is specified explicitly or the runtime environment generates the WSDL from the Java interface. The resulting XML of the response BytesMessage is deserialized using JAXB unmarshalling and returned to the client.

- **BytesMessage wire format that maps to single byte array messages**

Use the `wireFormat.jmsBytes` format to map between the payload of a BytesMessage and a `byte[]` argument or return value. To specify this wire format, add the following wire format element to the composition definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsBytes
```

OASIS

```
{http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsBytes
```

– **Mapping a request message payload to input arguments**

When using this wire format, the service interface method must contain a single parameter of type `byte[]`; for example: `void myMethod(byte[] bytes);`. The BytesMessage payload that is obtained using the `readBytes()` method is mapped to this single parameter.

– **Mapping an output return value to a response message payload**

The return type of the service method must be `byte[]` or `void`. If the return type is `byte[]`, the return value is set as the response BytesMessage payload.

– **Mapping exceptions to a response message payload**

When using this wire format, ObjectMessage messages are used to map exceptions, as in the ObjectMessage wire format case. If you have a checked exception, the runtime environment sets the exception as the ObjectMessage payload. For an unchecked exception, an instance of `RuntimeException` is set as the payload of a response ObjectMessage message. In both cases, a Boolean message property with name `org_apache_tuscany_sca_fault` is set to `true` on the response message.

- **JMS message wire format that does not extract the message payload and maps to the `javax.jms.Message` class**

You can use a wire format mechanism that instructs the JMS binding to provide the raw JMS message to the service implementation without extracting the payload out of the message. To use this wire format, the service interface must consist of a single method with one input parameter of `javax.jms.Message`, such as:

```
void methodA(javax.jms.Message msg);
```

The method name is not important; only the input parameter type is relevant. This method is similar to the `javax.jms.MessageListener` interface. The support for this method enables existing message-driven beans logic to be reused in SCA component implementations.

Use the default wire format to enable this mechanism to obtain the raw JMS message. To specify the default wire format, add the following wire format element to the composition definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsdefault
```

OASIS

```
{http://docs.oasis-open.org/ns/opencsa/sca/200912}wireFormat.jmsDefault
```

Because this wire format is the default, the same result is achieved by not specifying a wire format element.

This wire format is not the same as the TextMessage JAXB XML format. In this scenario, the wire format selection does not describe a specific format of data on the wire. Instead, the specification of this wire format indicates to the runtime environment to identify this specialty case to obtain the raw JMS message and handle the incoming data by passing the raw JMS message to the application.

The response message is a TextMessage with a `null` payload. The response message is sent unless the `onMessage` method is annotated as a one-way operation.

Message types for SCA references when the JMS binding wire format is predetermined

For an SCA reference to produce messages that are consumed in a predetermined JMS binding wire format, use the following message types:

- **TextMessage with serialized XML wire format that maps data using JAXB**

  The TextMessage wire format uses JAXB technology to marshal and unmarshal data into XML. To specify this wire format, add the following wire format element to the composite definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsTextXML
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsTextXML
  ```

  – **Mapping input arguments to a request message payload**

    Using this wire format, the input arguments are marshalled into an XML payload, and added to a request TextMessage. The format of the service is described by a WSDL file interface that is specified by either a `interface.wsdl` element from the component service definition, or by a WSDL that is generated from the Java interface of the SCA service if the `interface.wsdl` element is not explicitly defined. The Java-to-WSDL mapping is based on the JAXB and Java API for XML-Based Web Services (JAX-WS) specifications. This mapping results in a document-literal wrapped WSDL file, and the wrapper element corresponds to the operation name. JAXB marshalling is used to serialize the Java client input arguments into the XML payload.

  – **Mapping a response message payload to output return value and exceptions**

    The runtime environment first determines whether the response message represents an unchecked exception condition by checking for the `org_apache_tuscany_sca_fault` Boolean message property. If this property is set to `true` on the response message, the runtime environment expects an ObjectMessage as the response. When this condition occurs, a ServiceRuntimeException error is returned to the client. If this property is not set, or set to `false`, the message is a TextMessage that can represent either a checked exception or output data.

    The return value of the service method is unmarshalled in a similar manner to the marshalling of the request message input. The marshalling uses JAXB and JAX-WS technology to map to XML in a WSDL defined format such that the WSDL is specified explicitly or the runtime environment generates the WSDL from the Java interface. The XML payload of the response TextMessage message is deserialized using JAXB unmarshalling and returned to the client.

    If you have a checked exception, the TextMessage payload contains a fault in XML serialized form. The runtime environment deserializes the fault using JAXB unmarshalling and the corresponding checked exception is sent back to the client, wrappering the fault data. To learn more about exceptions and faults, see the documentation on using business exceptions with SCA interfaces.

- **The BytesMessage or TextMessage with serialized XML wire format that maps data using JAXB**

The BytesMessage or TextMessage wire format uses JAXB technology to marshal and unmarshal data into XML. This wire format is the default wire format. Thus, this wire format applies if no wire format element is specified in the composite definition file. To specify this wire format, add the following wire format element to the composite definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsdefault
```

OASIS

```
{http://docs.oasis-open.org/ns/opencsa/sca/200912}wireFormat.jmsDefault
```

– **Mapping input arguments to a request message payload**

  Using this wire format, the input arguments are marshalled into an XML payload, and added to a request BytesMessage. The format of the service is described by a WSDL file interface that is specified by either a `interface.wsdl` element from the component reference definition, or by a WSDL that is generated from the Java interface of the SCA service if an `interface.wsdl` element is not defined.

  The runtime Java-to-WSDL mapping is based on the JAXB and JAX-WS specifications. This mapping results in a document-literal wrapped WSDL file, and the wrapper element corresponds to the operation name, except when there is a single parameter method. A single argument input parameter is never wrapped by a wrapper element corresponding to the operation name. JAXB marshalling is used to serialize the Java client input arguments into the XML payload.

– **Mapping a response message payload to output return value and exceptions**

  The runtime environment first determines whether the response message represents an unchecked exception condition by checking for the `org_apache_tuscany_sca_fault` Boolean message property. If this property is set to `true` on the response message, the runtime environment expects an ObjectMessage as the response. When this condition occurs, a ServiceRuntimeException error is returned to the client. If this property is not set, or set to `false`, the message is a BytesMessage or TextMessage that can represent either a checked exception or output data.

  If you have a checked exception, the BytesMessage or TextMessage payload contains a fault in XML serialized form. The runtime environment deserializes the fault using JAXB unmarshalling and the corresponding checked exception is sent back to the client, wrappering the fault data. To learn more about exceptions and faults, see the topic on using business exceptions with SCA interfaces.

  For non-exception output data, the return value of the service method is unmarshalled in a similar manner to the marshalling of the request message input. The unmarshalling uses JAXB and JAX-WS technology to map to XML in a WSDL defined format such that the WSDL is specified explicitly or the runtime environment generates the WSDL from the Java interface. The XML payload is always expected to be in unwrapped format even if the WSDL defines a wrapper with an operation name. The XML payload of the response BytesMessage or TextMessage is deserialized using JAXB unmarshalling and returned to the client.

• **ObjectMessage wire format that maps to the `java.lang.Object` class**

  The ObjectMessage wire format uses serialized Java objects. To specify this wire format, add the following wire format element to the composition definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsObject
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsObject
  ```

  – **Mapping input arguments to a request message payload**

    The arguments that are passed to the SCA reference are wrapped in an array and are set into the request ObjectMessage payload using the `setObject()` method, except for methods with exactly one parameter. By default, arguments are not wrapped in an array for methods with a single argument unless the `wrapSingle` attribute is set to `true` on the `wireFormat.jmsObject` element.

  – **Mapping a response message payload to output return value and exceptions**

The runtime environment obtains the response ObjectMessage payload using the `getObject()` method. If the payload object is an unchecked exception, a ServiceRuntimeException error is returned to the client. If the payload object is an instance of a checked exception, the checked exception is returned to the client If the payload object represents non-exception output data, that payload object is returned back to the client application.

- **TextMessage wire format that maps to single string messages**

  Use the `wireFormat.jmsText` format to map between the payload of a TextMessage and a single `String` argument or return value. To specify this wire format, add the following wire format element to the composition definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsText
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsText
  ```

  – **Mapping input arguments to a request message payload**

    When using this wire format, the service interface method typically has a single parameter of type `String` ; for example: `void myMethod(String str)`. The single `String` passed to the SCA reference is set into the request TextMessage payload by the `setText()` method.

    For method signatures with different input parameters, the result of performing a `String.valueOf()` method on the first argument is set as the payload, regardless of how many arguments are present.

  – **Mapping a response message payload to output return value and exceptions**

    The return type of the service method is must be either `String` or `void`; for example: `String myMethod(String str)`.

    The runtime environment first determines whether the response message represents an exception condition by checking for the `org_apache_tuscany_sca_fault` Boolean message property. If this property is set to `true` on the response message, the runtime environment expects an ObjectMessage message as the response, and the message payload is obtained by the `getObject()` method. If the payload object is an unchecked exception, a ServiceRuntimeException is returned back to the client. If the payload object is an instance of a checked exception, that checked exception is returned to the client.

    If this property is not set, or set to `false`, the runtime environment expects non-exception output data and a TextMessage response. In this case, if the return type is `String`, a `getText()` method used to obtain the payload object that will be returned to the client application as a `String` value.

- **The BytesMessage with XML wire format that maps data using JAXB**

  The BytesMessage wire format uses JAXB technology to marshal and unmarshal data into XML. To specify this wire format, add the following wire format element to the composite definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsBytesXML
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsBytesXML
  ```

  – **Mapping input arguments to a request message payload**

    Using this wire format, the input arguments are marshalled into an XML payload, and added to a request BytesMessage. The format of the service is described by a WSDL file interface that is specified by either a `interface.wsdl` element from the component reference definition, or by a WSDL that is generated from the Java interface of the SCA service if an `interface.wsdl` element is not defined.

    The runtime Java-to-WSDL mapping is based on the JAXB and JAX-WS specifications. By default, this mapping results in a document-literal wrapped WSDL file, and the wrapper element corresponds to the operation name. JAXB marshalling is used to serialize the Java client input arguments into the XML payload.

  – **Mapping a response message payload to output return value and exceptions**

The runtime environment first determines whether the response message represents an unchecked exception condition by checking for the `org_apache_tuscany_sca_fault` Boolean message property. If this property is set to `true` on the response message, the runtime environment expects an ObjectMessage as the response. When this condition occurs, a ServiceRuntimeException error is returned to the client. If this property is not set, or set to `false`, the message is a BytesMessage that can represent either a checked exception or normal output data.

If you have a checked exception, the BytesMessage payload contains a fault in XML serialized form. The runtime environment deserializes the fault using JAXB unmarshalling and the corresponding checked exception is sent back to the client, wrappering the fault data. To learn more about exceptions and faults, see the topic on using business exceptions with SCA interfaces.

For non-exception output data, the return value of the service method is unmarshalled in a similar manner to the marshalling of the request message input. The unmarshalling uses JAXB and JAX-WS technology to map to XML in a WSDL defined format such that the WSDL is specified explicitly or the runtime environment generates the WSDL from the Java interface. The XML payload of the response BytesMessage is deserialized using JAXB unmarshalling and returned to the client.

- **BytesMessage wire format that maps to single byte array messages**

  Use the `wireFormat.jmsBytes` format to map between the payload of a BytesMessage and a single `byte[]` argument or return value. To specify this wire format, add the following wire format element to the composition definition file:

  OSOA

  ```
  {http://tuscany.apache.org/xmlns/sca/1.0}wireFormat.jmsBytes
  ```

  OASIS

  ```
  {http://tuscany.apache.org/xmlns/sca/1.1}wireFormat.jmsBytes
  ```

  – **Mapping input arguments to a request message payload**

    When using this wire format, the service interface method must contain a single parameter of type `byte[]`; for example: `void myMethod(byte[] bytes);`. The single argument is mapped to a BytesMessage payload using the `writeBytes()` method.

  – **Mapping a response message payload to output return value and exceptions**

    The return type of the service method must be `byte[]` or `void`.

    The runtime environment first determines whether the response message represents an exception condition by checking for the `org_apache_tuscany_sca_fault` Boolean message property. If this property is set to `true` on the response message, the runtime environment expects an ObjectMessage message as the response, and the message payload is obtained by the `getObject()` method. If the payload object is an unchecked exception, a ServiceRuntimeException error is returned to the client. If the payload object is an instance of a checked exception, that checked exception is returned to the client

    If this property is not set, or set to `false`, the runtime environment expects non-exception output data and a BytesMessage response message. In this case, a `readBytes()` method is used to obtain the payload object that will be returned to the client application as a `byte[]` value.

Message types for SCA services and references when the JMS binding wire format is not predetermined

Suppose you are adding a JMS binding to an SCA service or reference to produce messages that will be consumed at a future time by a JMS producer or consumer, and there is not a predetermined wire format.

It is a best practice to use the default wire format when starting with the SCA application. Use the JAXB programming model with the top-down approach to developing SCA applications as these service implementations and clients are easily used with the SCA default binding, the SCA web service binding, and the SCA JMS binding. Adopting an XML-centric view of your business data provides maximum portability across diverse platforms and technologies, and takes advantage of the design goals of a typical SOA environment. To learn more about top-down development of SCA applications, see the developing SCA applications from existing WSDL files documentation.

If you have business data that is described within Java classes that implement the Java serialization interface, `java.io.Serializable`, but the JAXB marshalling and unmarshalling does not satisfactorily preserve the data over the wire, you can use the ObjectMessage wire format.

Configuring JMS binding request and response wire formats:

You can configure the messaging data formats between an SCA application and a JMS producer or consumer, by configuring your SCA application to take advantage of a supported message type and data format. In general, each wire format can map to the service or reference side, and even into serialization and deserialization. As a result, you can configure each service or reference request and response to use different wire formats.

Before you begin

Configure the JMS binding for your SCA application. Then, configure the JMS binding wire format.

The product supports both the OASIS and the OSOA SCA JMS binding specification. Unless otherwise specified, the information in this topic pertains to applications for both the OSOA and OASIS specification.

About this task

In most cases, the response wire format can be the same as the request wire format for a messaging application. However, in certain scenarios this might not be reasonable, such as when the inputs and outputs of an operation cannot use the same wire format. In this situation, you can override the request wire format by explicitly configuring the response wire format with a `wireformat` element as a child on the `binding.jms response` element.

When choosing a request and response wire format, consider any restrictions imposed by the application itself and also the limitations of any particular wire format. For example, consider the following interface method:

```
public MyJavaObject method(MyJAXBObject mjo)
```

`MyJavaObject` is not an XML-serializable object and `MyJAXBObject` is not Java-serializable so you cannot use only one wire format. However, you can use `wireFormat.jmsTextXML` for the request wire format and `wireFormat.jmsObject` for the response.

Procedure

1. Open an editor on a composite definition file and configure a reference-side wire format.

   A reference-side wire format resembles the following:

   OSOA

```
<component name="JAXBJMSFrontendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.frontend.HelloWorldJAXBFrontendImpl"/>
    <reference name="hwJAXBService">
      <interface.java interface="com.ibm.test.soa.sca.HelloWorldJAXBService"/>
        <binding.jms>
          <destination name="jms/SCA_JMS_Request1"/>
          <connectionFactory name="jms/SCA_JMS_CF"/>
          <response>
            <destination name="jms/SCA_JMS_Response1"/>
            <connectionFactory name="jms/SCA_JMS_CF"/>
            <ts:wireFormat.jmsObject/>
          </response>
        </binding.jms>
    </reference>
</component>
```

   OASIS

```
<component name="JAXBJMSFrontendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.frontend.HelloWorldJAXBFrontendImpl"/>
    <reference name="hwJAXBService">
```

```
      <interface.java interface="com.ibm.test.soa.sca.HelloWorldJAXBService"/>
        <binding.jms>
          <destination jndiName="jms/SCA_JMS_Request1"/>
          <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          <response>
            <ts:wireFormat.jmsObject/>
            <destination jndiName="jms/SCA_JMS_Response1"/>
            <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          </response>
        </binding.jms>
    </reference>
</component>
```

In the example component configuration, the binding level wire format is the default because no wire format is specified. However, the response wire format is overridden by the `jmsObject` wire format.

2. Open an editor on a composite definition file and configure a service-side wire format.

   A service-side wire format resembles the following:

   OSOA

```
<component name="JAXBJMSBackendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.backend.HelloWorldJAXBBackendImpl"/>
    <service name="HelloWorldJAXBService">
      <interface.java interface="com.ibm.test.soa.sca.HelloWorldJAXBService"/>
        <binding.jms>
          <destination name="jms/SCA_JMS_Response1"/>
          <activationSpec name="jms/SCA_JMS_AS1"/>
          <response>
            <destination name="jms/SCA_JMS_Response1"/>
            <connectionFactory name="jms/SCA_JMS_CF"/>
            <ts:wireFormat.jmsObject/>
          </response>
        </binding.jms>
    </service>
</component>
```

   OASIS

```
<component name="JAXBJMSBackendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.backend.HelloWorldJAXBBackendImpl"/>
    <service name="HelloWorldJAXBService">
      <interface.java interface="com.ibm.test.soa.sca.HelloWorldJAXBService"/>
        <binding.jms>
          <destination jndiName="jms/SCA_JMS_Response1"/>
          <activationSpec jndiName="jms/SCA_JMS_AS1"/>
          <response>
            <ts:wireFormat.jmsObject/>
            <destination jndiName="jms/SCA_JMS_Response1"/>
            <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          </response>
        </binding.jms>
    </service>
</component>
```

   The response wire format is always the same as the request wire format unless the response level wire format is explicitly configured.

Results

You have configured the request and response wire format for a messaging application.

Examples

More example component configurations that show request and response wire formats follow.

Component configuration for a multiple-parameter reference

OSOA

```
<component name="MultipleParameterFrontendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.frontend.MultipleParameterFrontendImpl"/>
    <reference name="mpService">
      <interface.java interface="com.ibm.test.soa.sca.MultipleParameterService"/>
        <binding.jms>
          <destination name="jms/SCA_JMS_Request2"/>
          <connectionFactory name="jms/SCA_JMS_CF"/>
          <response>
            <destination name="jms/SCA_JMS_Response2"/>
            <connectionFactory name="jms/SCA_JMS_CF"/>
            <ts:wireFormat.jmsObject/>
          </response>
          <ts:wireFormat.jmsObject/>
        </binding.jms>
    </reference>
</component>
```

OASIS

```
<component name="MultipleParameterFrontendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.frontend.MultipleParameterFrontendImpl"/>
    <reference name="mpService">
      <interface.java interface="com.ibm.test.soa.sca.MultipleParameterService"/>
        <binding.jms>
          <ts:wireFormat.jmsObject/>
          <destination jndiName="jms/SCA_JMS_Request2"/>
          <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          <response>
            <ts:wireFormat.jmsObject/>
            <destination jndiName="jms/SCA_JMS_Response2"/>
            <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          </response>
        </binding.jms>
    </reference>
</component>
```

Component configuration for a multiple-parameter service that uses `jmsBytesXML` wire format

OSOA

```
<component name="MultiParameterJMSBackendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.backend.MultipleParameterBackendImpl"/>
    <service name="MultipleParameterService">
      <interface.java interface="com.ibm.test.soa.sca.MultipleParameterService"/>
        <binding.jms>
          <destination name="jms/SCA_JMS_Request2"/>
          <activationSpec name="jms/SCA_JMS_AS2"/>
          <response>
            <destination name="jms/SCA_JMS_Response2"/>
            <connectionFactory name="jms/SCA_JMS_CF" create="never"/>
            <ts:wireFormat.jmsBytesXML/>
          </response>
        </binding.jms>
    </reference>
</component>
```

OASIS

```
<component name="MultiParameterJMSBackendReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.backend.MultipleParameterBackendImpl"/>
    <service name="MultipleParameterService">
      <interface.java interface="com.ibm.test.soa.sca.MultipleParameterService"/>
        <binding.jms>
          <destination jndiName="jms/SCA_JMS_Request2"/>
          <activationSpec jndiName="jms/SCA_JMS_AS2"/>
          <response>
            <ts:wireFormat.jmsBytesXML/>
            <destination jndiName="jms/SCA_JMS_Response2"/>
            <connectionFactory jndiName="jms/SCA_JMS_CF" create="never"/>
```

```
          </response>
        </binding.jms>
    </reference>
</component>
```

Component configuration for a reference that uses `jmsText` wire format for the response and a `jmsCustom` format for the binding

OSOA

```
<component name="FrontEndWireFormatHandlerReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.frontend.CustomReqRespFrontEndComponent"/>
    <reference name="frontEnd">
        <binding.jms>
          <destination name="jms/SCA_JMS_Request3"/>
          <connectionFactory name="jms/SCA_JMS_CF"/>
          <response>
            <destination name="jms/SCA_JMS_Response3"/>
            <connectionFactory name="jms/SCA_JMS_CF"/>
            <ts:wireFormat.jmsText/>
          </response>
          <ts:wireFormat.jmsCustom class="com.ibm.test.soa.sca.frontend.custom.FrontEndReqRespWireFormatHandler"/>
        </binding.jms>
    </reference>
</component>
```

OASIS

```
<component name="FrontEndWireFormatHandlerReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.frontend.CustomReqRespFrontEndComponent"/>
    <reference name="frontEnd">
        <binding.jms>
          <ts:wireFormat.jmsCustom class="com.ibm.test.soa.sca.frontend.custom.FrontEndReqRespWireFormatHandler"/>
          <destination jndiName="jms/SCA_JMS_Request3"/>
          <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          <response>
            <ts:wireFormat.jmsText/>
            <destination jndiName="jms/SCA_JMS_Response3"/>
            <connectionFactory jndiName="jms/SCA_JMS_CF"/>
          </response>
        </binding.jms>
    </reference>
</component>
```

Component configuration for a service that uses `jmsText` wire format for the response and a `jmsCustom` format for the binding

OSOA

```
<component name="BackEndWireFormatHandlerReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.backend.CustomReqRespBackEndComponent"/>
    <service name="CustomReqResp">
        <binding.jms>
          <destination name="jms/SCA_JMS_Request3" type="queue" create="never"/>
          <activationSpec name="jms/SCA_JMS_AS3"/>
          <response>
            <destination name="jms/SCA_JMS_Response3"/>
            <connectionFactory name="jms/SCA_JMS_CF" create="never"/>
            <ts:wireFormat.jmsText/>
          </response>
          <ts:wireFormat.jmsCustom class="com.ibm.test.soa.sca.backend.custom.BackendReqRespDBH"/>
        </binding.jms>
    </service>
</component>
```

OASIS

```
<component name="BackEndWireFormatHandlerReqRespWFComponent">
  <implementation.java class="com.ibm.test.soa.sca.backend.CustomReqRespBackEndComponent"/>
    <service name="CustomReqResp">
```

```
        <binding.jms>
          <ts:wireFormat.jmsCustom class="com.ibm.test.soa.sca.backend.custom.BackendReqRespDBH"/>
          <destination jndiName="jms/SCA_JMS_Request3" type="queue" create="never"/>
          <activationSpec jndiName="jms/SCA_JMS_AS3"/>
          <response>
            <ts:wireFormat.jmsText/>
            <destination jndiName="jms/SCA_JMS_Response3"/>
            <connectionFactory jndiName="jms/SCA_JMS_CF" create="never"/>
          </response>
        </binding.jms>
    </service>
</component>
```

What to do next

Deploy and test the wire format configuration in your SCA application.

Configuring transactions for the SCA JMS binding

You can configure the Service Component Architecture (SCA) Java Message Service (JMS) binding for services and references to take advantage of transaction quality of service behaviors.

Before you begin

The information in this topic applies for both OSOA and OASIS SCA applications.

About this task

SCA provides declarative mechanisms in the form of intents for describing the transactional environment required by components.

The SCA JMS binding supports transacting message delivery with the global transaction of a component. The SCA transaction policies are specified as intents that represent quality of service behavior offered by the JMS binding on an SCA service or reference. However, the SCA JMS binding does not propagate transaction context; therefore, the client and service cannot participate in the same global transaction.

To learn more about SCA global transactions, see the SCA transaction intents documentation.

SCA references can use the `transactedOneWay` intent to transact one-way requests. When the `transactedOneWay` intent is used on an SCA reference, a one-way request on the reference is not sent until the global transaction of the client is committed. If the global transaction of the client is rolled back, the request is not sent.

SCA services can use the `transactedOneWay` intent to transact one-way requests only or the `exactlyOnce` intent to transact both one-way and request-response message patterns. When the `transactedOneWay` intent is used on an SCA service, a one-way request is received from the JMS binding as part of the component's global transaction. When the `exactlyOnce` intent is used on a service, both one-way and request-response message patterns are received from the JMS binding as part of the component's global transaction. The receipt of the message, and the sending of the response for request-response messaging, is not committed until the service transaction commits. If the service transaction is rolled back, the message is again made available for delivery or the message is sent to an exception destination that is based upon the configuration of the bus destination.

The SCA runtime environment typically performs a rollback of a global transaction only if the component produces an unchecked exception error. An unchecked exception error is a subclass of java.lang.RuntimeException or java.lang.Error class. A checked exception does not force a rollback. The component can force a rollback by using the UOWSynchronizationRegistry interface. For example:

```
com.ibm.websphere.uow.UOWSynchronizationRegistry uowSyncRegistry =
    com.ibm.wsspi.uow.UOWManagerFactory.getUOWManager();
uowSyncRegistry.setRollbackOnly();
```

If a reference does not require a `transactedOneWay` intent, then one-way requests are sent immediately. If a service does not require the `transactedOneWay` or `exactlyOnce` intent, requests are removed from the queue prior to the delivery of the request to the component. If the component processing fails, these requests cannot be delivered again.

Procedure

1. Specify the `transactedOneWay` or `exactlyOnce` intents on your SCA service or reference enable transacting message delivery with the global transaction of your component.

   The following example illustrates the use of the `transactedOneWay` and `exactlyOnce` intents. In this example, the component, `TransactionalComponent`, receives one-way and request-response messages from the DataUpdate service and subsequently sends one-way requests to the loggingService reference transactionally. If the component transaction rolls back, the active request is queued again and any requests to the reference are not sent.

   ```
   <component name="TransactionalComponent">
        <implementation.java class="example.TransactedImpl"
                requires="managedTransaction.global"/>
        <service name="DataUpdate" requires="exactlyOnce">
            <binding.jms>
               <destination name="jms/DataUpdate_Request" type="queue"/>
               <activationSpec name="jms/SCA_JMS_AS"/>
            </binding.jms>
        </service>

        <reference name="loggingService" requires="transactedOneWay">
            <binding.jms>
               <connectionFactory name="jms/SCA_JMS_CF"/>
            </binding.jms>
        </reference>
    </component>
   ```

2. Using the administrative console, configure the bus destination to handle failed messages.
   a. Start the administrative console.
   b. In the navigation pane, click **Service integration** > **Buses** > ***bus_name*** > **Destinations** > ***destination_queue_name*** or ***destination_topic_space_name*** .
   c. Under Exception destination, enter a value for the **Maximum failed deliveries per message** . This value specifies the maximum number of failed attempts to process a message, after which the message is forwarded from its intended destination to the exception destination. This property applies to individual messages.

Results

You have updated your SCA composite to use transacted message delivery.

Dynamic JMS resource creation during deployment

The product dynamically creates Java Message Service (JMS) resources necessary for a Service Component Architecture (SCA) composite, if those resources do not exist and relate to the WebSphere default messaging provider. The product creates the resources when adding the SCA composite to a business-level application. The dynamically created resources are created in a WebSphere default messaging provider service integration bus. The product does not create resources that relate to WebSphere MQ; those resources must exist. A dynamically created service integration bus resource is given a name that is specified in the JMS binding or, if the binding does not specify a resource name, is given a default name. When an SCA composite uses a mixture of existing and non-existent resources, the product dynamically creates the resources that do not exist.

Dynamic resource creation is supported for applications coded to both the OSOA and OASIS specifications. Unless otherwise specified, the information in this topic pertains to both OSOA and OASIS applications.

Restriction: Dynamic resource creation is not supported for multiple-server configurations. For stand-alone application servers, dynamic resource creation is enabled by default. To disable dynamic resource creation, set the `admin.jms.DRC.disable` system property to `true`.

- Default naming of resources
- Deployment validation error or warning messages
- Dynamic resource creation for JMS bindings

Default naming of resources

When a JMS binding does not specify a Java Naming and Directory Interface (JNDI) name for a resource, the product dynamically creates a service integration bus resource and assigns the resource a default name, `DefaultSCABus`.

Restriction: You must have SIBus service enabled for your application server. Before deploying your SCA composite, enable the SIBus service of the server, and then stop and restart the server.

For an SCA service that uses a JMS binding, the product uses the following default names if JNDI name values are not supplied in the composite definition.

Table 88. Default names for service resources. The product uses the default names when the composite definition omits JNDI name values.

Resource	Default name
Activation specification	`jms/DefaultSCAActivationSpec`
Activation specification create	OSOA: `ifnotexist` OASIS: `ifNotExist`
Destination	`jms/<componentName>_<serviceName>_ServiceRequestDestination`
Response connection factory name	`jms/DefaultSCAConnectionFactory`
Response connection factory create	OSOA: `ifnotexist` OASIS: `ifNotExist`

The product assigns computed destination names for values that are not provided in the JMS binding element based on the component name and service name, separated with underscores. For example, the computed destination name for a JMS binding in the HelloService service of the HelloServiceComponent component is `jms/HelloServiceComponent_HelloService_ServiceRequestDestination`.

The product assigns the destination a default value only when no JNDI name is supplied in the composite definition and the default activation specification is being used. If the composite definition defines an activation specification that exists already, the destination from the activation specification is used instead of the default value.

For an SCA reference that uses a JMS Binding, the product uses the following default names if JNDI name values are not supplied in the composite definition.

Table 89. Default names for reference resources. The product uses the default names when the composite definition omits JNDI name values.

Resource	Default name
Connection factory	`jms/DefaultSCAConnectionFactory`
Connection factory create	OSOA: `ifnotexist` OASIS: `ifNotExist`

Table 89. Default names for reference resources (continued). The product uses the default names when the composite definition omits JNDI name values.

Resource	Default name
Response connection factory	`jms/DefaultSCAConnectionFactory`
Response connection factory create	OSOA: `ifnotexist` OASIS: `ifNotExist`

The product assigns the response connection factory a default value only when no JNDI name is supplied in the composite definition and the response destination has been defined.

Deployment validation error or warning messages

The product validates a composite definition when adding an SCA asset to a business-level application. If the validation results in an error, the product does not add the asset to the application. If a warning results, the product issues a warning but adds the asset to the application.

Errors or warnings from validation of service JMS bindings

When a composite definition sets a `create` attribute to `always` for any of the following service JMS bindings, deployment stops with an error if the resource exists:
- Destination
- Activation specification
- Response destination
- Response connection factory

When a composite definition sets a `create` attribute to `never` for any resource, and that resource does not exist, the product issues a warning but the deployment continues and can complete successfully.

Errors or warnings from validation of reference JMS bindings

When a composite definition does not define a callback service or `destination` attribute, deployment stops with an error.

When a composite definition sets a `create` attribute to `always` for any of the following reference JMS bindings, deployment stops with an error if the resource exists:
- Destination
- Connection factory
- Response destination
- Response connection factory

When a composite definition sets a `create` attribute to `never` for any resource, and that resource does not exist, the product issues a warning but the deployment continues and can complete successfully.

An error occurs when an activation specification or connection factory has a default value assigned, and a non-default bus is used. This error can happen if the composite definition has an existing resource defined, such as a destination, and it uses a non-default bus. If the activation specification or connection factory has not been defined, the product supplies the default name.

Dynamic resource creation for JMS bindings

The product follows patterns for dynamic resource creation that are similar, although different, for services and references.
- Service resource creation
- Reference resource creation

Service resource creation:

When a service JMS binding is deployed, the product attempts to create resources for the binding. The composite definition might not specify JNDI names for the resources. The resources might not exist. There are four typical patterns for dynamic service resource creation:

Composite definition does not define resources

A composite definition that does not define resources resembles:

```
<binding.jms>
</binding.jms>
```

If a composite definition does not define resources, the product does the following:

- Assigns default names to all resources using the default service integration bus.
- Creates the default service integration bus `DefaultSCABus` if it does not exist.
- Creates the default destination `jms/<componentName>_<serviceName>_ServiceRequestDestination` on the default bus.
- Creates the default activation specification `jms/DefaultSCAActivationSpec` if it does not exist and uses the default bus and default destination.
- Creates the default response connection factory `jms/DefaultSCAConnectionFactory` if it does not exist and uses the default bus.

When the product creates a destination, a bus destination and a destination resource are created. The bus destination name and the destination resource name are derived from the JNDI name, with / replaced by _.

Composite definition defines destination and activation specification

A composite definition that defines both a destination and an activation specification resembles:

OSOA

```
<binding.jms>
  <destination name="jms/myDestination_Request" type="queue"/>
  <activationSpec name="jms/myActivationSpec"/>
</binding.jms>
```

OASIS

```
<binding.jms>
  <destination jndiName="jms/myDestination_Request" type="queue"/>
  <activationSpec jndiName="jms/myActivationSpec"/>
</binding.jms>
```

If a composite definition defines a destination and activation specification, the product performs actions that depend on whether resources exist for the destination and activation specification.

Resources for both the destination and activation specification do not exist:

- Creates the default service integration bus `DefaultSCABus` if it does not exist.
- Creates the destination `jms/myDestinationRequest` on the default bus.
- Creates the activation specification `jms/myActivationSpec` if it does not exist and uses the default bus and default destination.

The destination resource does not exist, but the activation specification resource exists:

- Creates the destination `jms/myDestinationRequest` on the bus that the activation specification uses.

The destination resource exists, but the activation specification resource does not exist:

- Creates the activation specification `jms/myActivationSpec` with the bus that the destination uses.

Composite definition defines the destination only

A composite definition that defines a destination resembles:

OSOA

```
<binding.jms>
  <destination name="jms/myDestination_Request" type="queue"/>
</binding.jms>
```

OASIS

```
<binding.jms>
  <destination jndiName="jms/myDestination_Request" type="queue"/>
</binding.jms>
```

If a composite definition defines a destination, the product performs actions that depend on whether the resource exists for the destination.

Destination resource does not exist:

- Creates the default service integration bus `DefaultSCABus` if it does not exist.
- Creates the destination `jms/`*`myDestinationRequest`* on the default bus.
- Creates the default activation specification `jms/DefaultSCAActivationSpec` if it does not exist and uses the default bus and the destination `jms/`*`myDestinationRequest`*.

Destination resource exists:

- The destination must use the default bus. Otherwise, the product returns an error and does not add the asset to the business-level application.
- Creates the default activation specification `jms/DefaultSCAActivationSpec` if it does not exist and uses the default bus and the destination `jms/`*`myDestinationRequest`*.

Composite definition defines the activation specification only

A composite definition that defines an activation specification resembles:

OSOA

```
<binding.jms>
  <activationSpec name="jms/myActivationSpec"/>
</binding.jms>
```

OASIS

```
<binding.jms>
  <activationSpec jndiName="jms/myActivationSpec"/>
</binding.jms>
```

If a composite definition defines an activation specification, the product performs actions that depend on whether a resource exists for the activation specification.

Activation specification resource does not exist:

- Creates the default service integration bus `DefaultSCABus` if it does not exist.
- Creates the default destination `jms/`*`<componentName>`*`_`*`<serviceName>`*`_ServiceRequestDestination` on the default bus.
- Creates the activation specification `jms/`*`myActivationSpec`* and uses the default bus and default destination.

Activation specification resource exists:

- Does not create a destination. The run time uses the destination from the existing activation specification.

Response connection factory for JMS bindings that provide a two-way service

JMS bindings that provide a two-way service require a response connection factory. When an asset with a two-way service is added to a business-level application, the product behaves in the following manner:

Composite definition defines the response connection factory

- Creates the response connection factory if it does not exist.

- If any resources for the binding already exist, creates the response connection factory using the bus that the existing resource uses. Otherwise, the product uses the default bus.
- If other resources for the binding do not exist, creates the other resources using the bus that an existing response connection factory uses.

Composite definition does not define the response connection factory

- Assigns the default name `jms/DefaultSCAConnectionFactory` and creates the response connection factory if it does not exist.
- If the product creates a response connection factory using the default name, the response connection factory can only be created on the default bus. Otherwise, the product returns an error and does not add the asset to the business-level application.

  If this error occurs, specify a JNDI name for the response connection factory in the composite definition to correct the error.

Reference resource creation:

When a reference JMS binding is deployed, the product attempts to create resources for the binding. The composite definition must define a destination. The connection factory, response destination, and response connection factory might not be defined. The resources that are defined might not exist. There are four typical patterns for dynamic reference resource creation:

Composite definition defines the destination only

A composite definition that defines a destination resembles:

OSOA

```
<binding.jms>
  <destination name="jms/myDestination_Request" type="queue"/>
</binding.jms>
```

OASIS

```
<binding.jms>
  <destination jndiName="jms/myDestination_Request" type="queue"/>
</binding.jms>
```

The destination resource might not exist.

Destination resource does not exist:

- Creates the default SIBus `DefaultSCABus` if it does not exist.
- Creates the destination `jms/`*`myDestination_Request`* on the default bus.
- Creates the default connection factory `jms/DefaultSCAConnectionFactory` if it does not exist and uses the default bus.

Destination resource exists:

- Creates the default service integration bus `DefaultSCABus` if it does not exist.
- Creates the default connection factory `jms/DefaultSCAConnectionFactory` if it does not exist and uses the default bus.

Composite definition defines destination and connection factory

A composite definition that defines both a destination and a connection factory resembles:

OSOA

```
<binding.jms>
  <destination name="jms/myDestination_Request" type="queue"/>
  <connectionFactory name="jms/myConnectionFactory"/>
</binding.jms>
```

OASIS

```
<binding.jms>
  <destination jndiName="jms/myDestination_Request" type="queue"/>
  <connectionFactory jndiName="jms/myConnectionFactory"/>
</binding.jms>
```

Each of these resources might not exist.

Resources for both the destination and connection factory do not exist:

- Creates the default service integration bus `DefaultSCABus` if it does not exist.
- Creates the destination `jms/`*`myDestination_Request`* on the default bus.
- Creates the connection factory `jms/myConnectionFactory` if it does not exist and uses the default bus and default destination.

The destination resource exists, but the connection factory resource does not exist:

- Creates the connection factory `jms/myConnectionFactory` with the bus that the destination uses.

The destination resource does not exist, but the connection factory resource exists:

- Creates the destination `jms/`*`myDestination_Request`* on the bus that the connection factory uses.

Invoking operations using JMS binding operation selection

You can invoke an operation using JMS binding operation selection. By default, a JMS binding uses the JMS binding operation selection.

Before you begin

Configure the JMS binding for your SCA application.

About this task

The JMS binding operation selection is the default binding selection. Thus, this operation selection applies if no operation selection element is specified in the composite definition file. To specify the JMS binding operation selection, add the following operation selection element to the composite definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}operationSelector.jmsdefault
```

OASIS

```
{http://docs.oasis-open.org/ns/opencsa/sca/200912}operationSelector.jmsDefault
```

Specify a JMS binding operation selection on an SCA service interface and on an SCA reference.

Procedure

1. In a composite definition file, specify a JMS binding operation selection on an SCA service interface.

   Because a given service can define multiple operations, the runtime environment must define mechanisms and algorithms to take a message from a destination bound from an SCA service using the SCA JMS binding, and invoke the correct operation on the SCA service implementation.

   The JMS binding defines a `String` message property called `scaOperationName`. When receiving a request at a service, or a callback at a reference, the JMS binding uses the following algorithm to determine the operation name:

   a. If there is only one operation on the service interface, it is assumed that this operation is the operation name for the request.
   b. Otherwise, if the JMS property `scaOperationName` is set, the value of this property is used as the operation name.

c. Otherwise, if the message is a JMS text or bytes message containing XML, then the selected operation name is taken from the local name of the root element of the XML payload. This operation selection behavior is only supported over the jmsdefault and the jmsTextXML and jmsBytesXML wire formats.

d. Otherwise, it is assumed that the operation name is `onMessage`.

2. In a composite definition file, specify a JMS user property operation selection on an SCA reference.

   When using the JMS user operation selector on the service side, the reference side or service side callback must set the matching JMS user property. In the service-side example, the operation selector uses the value of the `jmsTestUserProp` to determine the target operation. You can set this property for the reference or callback side by specifying one or more header properties in the `operationProperties` element; for example:

```
<operationProperties name="proxyMethodName">
<headers>
<property name="jmsTestUserProp">remoteMethodName</property>
</headers>
</operationProperties>
```

   In the example, a call to the `proxyMethodName` on the reference side sets the `jmsTestUserProp` user property to `remoteMethodName`. The `proxyMethodName` target operation is set to `remoteMethodName` when the service-side operation selector is configured as in the previous step.

Results

You have configured a JMS binding operation selection.

What to do next

Deploy and test the operation selection in your SCA application.

Invoking operations using JMS user property operation selection

You can invoke an operation using JMS user property operation selection.

Before you begin

Configure the JMS binding for your SCA application.

About this task

The JMS user property operation selection is a predefined operation selector that determines the target operation from the value of a given JMS user property. You can define the name of the JMS user property using the `propertyName` attribute of the `operationSelector.jmsUserProp` element. To specify the JMS user property operation selection, add the following operation selection element to the composite definition file:

OSOA

```
{http://tuscany.apache.org/xmlns/sca/1.0}operationSelector.jmsUserProp
```

OASIS

```
{http://tuscany.apache.org/xmlns/sca/1.1}operationSelector.jmsUserProp
```

Specify a JMS user property operation selection on an SCA service interface and on an SCA reference.

Procedure

1. In a composite definition file, specify a JMS user property operation selection on an SCA service interface.

   On the service side, the `jmsUserProp` operation selection uses the value of the given JMS user property to determine the target operation. For example:

```
<binding.jms>

...
    <ts:operationSelector.jmsUserProp propertyName="jmsTestUserProp"/>
...
</binding.jms>
```

The operation selector requires that you set a JMS user property to the target operation name. If the property is not set or there is no matching operation, the operation selector does not default to any other value, unlike the default JMS binding operation selector.

2. In a composite definition file, specify a JMS user property operation selection on an SCA reference.

   When using the JMS user operation selector on the service side, the reference side or service side callback must set the matching JMS user property. In the service-side example, the operation selector uses the value of the `jmsTestUserProp` to determine the target operation. You can set this property for the reference or callback side by specifying one or more header properties in the `operationProperties` element; for example:

```
<operationProperties name="proxyMethodName">
  <headers>
    <property name="jmsTestUserProp">remoteMethodName</property>
  </headers>
</operationProperties>
```

   In the example, a call to the `proxyMethodName` on the reference side sets the `jmsTestUserProp` user property to `remoteMethodName`. The `proxyMethodName` target operation is set to `remoteMethodName` when the service-side operation selector is configured as in the previous section.

Results

You have configured a JMS user property operation selection.

What to do next

Deploy and test the operation selection in your SCA application.

Invoking operations using custom operation selectors

If your application cannot use a default JMS binding operation selection or JMS user property operation selection to determine the target operation, you can use a JMS custom operation selector to invoke operations.

Before you begin

Configure the JMS binding for your SCA application.

It is recommended that your SCA component implementations contain only business logic to improve their portability, such as over other bindings. To improve portability, use the custom operation selector to hold the JMS-specific logic that handles an incoming JMS Message, enabling the component implementation to focus on binding-neutral business logic.

About this task

You can configure a JMS custom operation selector in an SCA composite definition without any changes to the application. To specify the JMS custom operation selection, add the following operation selection element to the composite definition file:

OSOA

```
{http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06}operationSelector.jmsCustom
```

OASIS

```
{http://www.ibm.com/xmlns/prod/websphere/sca/1.1}operationSelector.jmsCustom
```

The custom operation selector must implement the following interface:

```
package com.ibm.websphere.soa.sca.operationselector.jms.OperationSelector;
import javax.jms.Message;

public interface OperationSelector {
  public String getOperationName(Message msg);
  public WireFormatContext getWireFormatContext();
  public void setWireFormatContext(WireFormatContext ctx);
```

The interface exposes an instance of javax.jms.Message and enables interaction with the JMS user properties and message body. After doing operation selection, you do not need to use the `reset()` method to reposition the byte stream cursor for a BytesMessage. The run time automatically resets the cursor in all cases so that the wire format handler is not impacted by the operation selector reading the message first.

If the message body of a BytesMessage is read using any of the `read` methods available on a BytesMessage, reposition the byte stream to the beginning using the `reset()` method so that message processors can read the data in the entire message body intended for the targeted operation.

The wire format context is held by an instance of the com.ibm.websphere.soa.sca.wireformat.WireFormatContext class, which is passed with each invocation. The WireFormatContext class provides a java.util.Map interface where you can set property key and value pairs in the context. The WireFormatContext interface also provides several methods to extract useful information about the current context such as component and service names, invocation types, and the ability to mark exceptions. Refer to the Java documentation for the com.ibm.websphere.soa.sca.wireformat.WireFormatContext interface for a complete list of methods.

Procedure

1. In a composite definition file, configure a custom operation selector.

   Use the `operationSelector.jmsCustom` element and the `class` attribute to identify the customer operation selection implementation class. The custom operation selector applies when receiving a request at a service, or a callback at a reference.

   OSOA

   ```
   <component name="AccountComponent">
     <implementation.java ...>
     <service name="AccountService">
       <binding.jms>
         <destination ...>
         <response>
         ...
         </response>
         <xmlns:fep="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
                   fep:operationSelector.jmsCustom
                   class="com.ibm.test.soa.sca.opsel.SimpleCustomOpSel"/>
       </binding.jms>
     </service>
   </component>
   ```

   OASIS

   ```
   <component name="AccountComponent">
     <implementation.java ...>
     <service name="AccountService">
       <binding.jms>
         <xmlns:fep="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"
                   fep:operationSelector.jmsCustom
                   class="com.ibm.test.soa.sca.opsel.SimpleCustomOpSel"/>
         <destination ...>
         <response>
         ...
         </response>
       </binding.jms>
     </service>
   </component>
   ```

2. Write code that implements the JMS custom operation selector class.
   For the SimpleCustomOpSel example class, you can use code such as the following:

```
package com.ibm.test.soa.sca.opsel;

import com.ibm.websphere.soa.sca.operationselector.jms.OperationSelector.*;
import javax.jms.*;

public class SimpleCustomOpSel implements OperationSelector {

    private WireFormatContext opSelContext;

    @Override
    public String getOperationName(Message msg) {
        String retVal = computeOperationName(msg);
        return retVal;
    }

    @Override
    public WireFormatContext getWireFormatContext() {
        return opSelContext;
    }

    @Override
    public void setWireFormatContext(WireFormatContext ctx) {
        this.opSelContext = ctx;
    }

    /*
     * There is a pre-packaged operationSelector that can do this,
     * <ts:operationSelector.jmsUser>.
     */
     private String computeOperationName(Message msg) {
         String opName = null;
         Integer val = null;
         try {
             val = msg.getIntProperty("MyPropertyName");
             if (val.equals(Integer.valueOf("1"))) {
                 opName = "addOperation";
             } else if (val.equals(Integer.valueOf("2"))) {
...
```

Results

You have configured a custom operation selector.

What to do next

Package the operation selection implementation class with your application. The class must be loadable by an application-level class loader.

Deploy and test the operation selector in your SCA application.

Using Atom bindings in SCA applications

You can use an Atom binding in a Service Component Architecture (SCA) application to expose collections of data as an Atom feed or to reference existing external Atom feeds.

Before you begin

If you are unfamiliar with the Atom protocol, refer to documentation on the Atom Syndication Format, an XML-based document format that describes web feeds, and the Atom Publishing Protocol, a protocol for publishing and updating web resources.

About this task

Use the Atom binding to work with services that provide or consume entries described in the Atom Syndication Format and Atom Publishing Protocol. An SCA component can reference existing external web feeds defined using the Atom protocol and work with them inside a Java implementation. Also, you can use the Atom binding to compose new services and expose them as an Atom feed.

This topic describes the following procedures:

- Expose an Atom feed service using an Atom binding.
- Use an Atom binding reference to access services exposed by SCA applications using the Atom binding or to access external Atom feeds.

Procedure

- Expose an Atom feed service using an Atom binding.
  1. Configure the Atom feed service in an SCA composite definition.

     Specify the uniform resource identifier (URI) of the Atom feed in a service in the composite definition of an SCA composite. The following example of a composite definition has a service exposed over the Atom binding:

```
<component name="NewsServiceComponent">
  <implementation.java class="com.ibm.test.atom.NewsServiceImpl"/>
  <service name="NewsService">
    <t:binding.atom uri="http://localhost:9080/newsService"/>
  </service>
</component>
```

     The example Atom binding URI, `http://localhost:9080/newsService`, is an absolute URI. To run applications that use an Atom binding in product clusters, specify a relative URI; for example:

```
<t:binding.atom uri="/newsService"/>
```

  2. Access the service.

     For example, to access the `NewsService` service, either use an SCA reference in another component or directly access the URI `http://localhost:9080/newsService` from a web browser. If accessed from a web browser, the browser handles the output as an Atom feed.

     The following example shows XML tagging for an Atom feed returned from the `NewsService` service:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Feed</title>
  <id>Feed1329090360</id>
  <entry>
    <id>Item1</id>
    <title type="text">item</title>
    <content type="application/xml">
      <ns2:root xmlns:ns2="http://tuscany.apache.org/xmlns/sca/databinding/jaxb/1.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="item">
      <name xmlns="">First Item Name</name>
      <title xmlns="">First Item Title</title>
    </ns2:root>
  </content>
  <link href="Item1"></link>
 </entry>
</feed>
```

  3. Manipulate the Atom feed using HTTP actions.

     You can manipulate the collections of items exposed by an Atom service by using the four basic actions of the HTTP protocol:
     - POST (create or add)
     - GET (retrieve or query)
     - PUT (update)
     - DELETE (remove)

     The Java implementation class for the component should implement the org.apache.tuscany.sca.data.collection.Collection interface or an interface that extends the Collection interface. The following methods are defined in the org.apache.tuscany.sca.data.collection.Collection interface:

public abstract Entry[] getAll()
Returns every entry in the feed
public abstract Entry[] query(String s)
Queries the feed for entries based on some matching criteria
public abstract Object post(Object obj, Object obj1)
Adds a new entry to the feed
public abstract Object get(Object obj) throws NotFoundException
Returns a single entry identified by the provided key value.
public abstract void put(Object obj, Object obj1) throws NotFoundException
Updates an existing entry identified by the provided key value
public abstract void delete(Object obj) throws NotFoundException
Deletes an existing entry identified by the provided key value

- Use an Atom binding reference to access services exposed by SCA applications using the Atom binding or to access external Atom feeds.
  1. Configure an Atom binding reference to access an exposed service.

     Specify a reference in the composite definition of an SCA composite that accesses an exposed service. The following example is a reference definition that accesses the `NewsService` service:

```
<reference name="newsServiceRef" target="NewsServiceComponent/NewsService">
    <t:binding.atom/>
</reference>
```

  2. Configure an Atom binding reference to access an external Atom feed.

     Specify a reference in the composite definition of an SCA composite that accesses an external Atom feed, for example:

```
<reference name="atomFeed">
    <tuscany:binding.atom uri="http://feeds.feedburner.com/blogspot/Dcni?format=xml"/>
</reference>
```

  3. Work with the Atom binding reference inside a Java implementation.

     A Java implementation class for an SCA component that contains the two example references might define them as follows:

```
@Reference(required = false)
public Collection newsServiceRef;

@Reference(required = false)
public Collection atomFeed;
```

  4. Use the Collection API in the implementation to manipulate the feed.

     For example, to add a new entry to the `NewsService` feed, the implementation might call:

```
MyEntry entry = new MyEntry("Title", "Content");
newsServiceRef.post("mykey-10-15", entry);
```

     To retrieve an entry from the external feed, the implementation code might use the following example code:

```
atomFeed.get("idtag-20090321");
```

What to do next

Deploy your SCA component in an application.

If the Atom feed service is deployed to a cluster and the target attribute, @target, is used to point to the service, the target URI resolves to an HTTP port for an individual cluster member. To maintain failover and load balancing in this situation, you can use an absolute URI on the `binding.atom` element that points to a proxy server endpoint rather than the target attribute on the reference element. For more information, see the topics on resolving SCA references and on routing HTTP requests to an SCA service when using an external web server.

Securing data exposed by Atom bindings

You can secure collections of data that are exposed by an Atom binding in a Service Component Architecture (SCA) application. An Atom binding can expose data as an Atom feed or reference existing external Atom feeds.

Before you begin

If you are unfamiliar with the Atom protocol, refer to documentation on the Atom Syndication Format, an XML-based document format that describes web feeds, and the Atom Publishing Protocol, a protocol for publishing and updating web resources.

For information about using Atom bindings in this product, refer to “Using Atom bindings in SCA applications” on page 806.

transition: In Version 8.5, the default value for the web authentication property `webAuthReq` is `persisting`, which enables credential persistence that allows login information to be available to unprotected web clients and enables additional access to user information. You must set the `webAuthReq` property to `lazy` to prevent unprotected web clients additional access to user information through persisting credentials. For more information, see Security hardening features enablement and migration. You can set `webAuthReq` to lazy on the administrative console Web security - General settings page:

1. Click **Security** > **Global security** > **Authentication** > **web and SIP security** > **General settings**.
2. Select **Authenticate only when the URI is protected**, which enables lazy authentication.
3. Click **Apply**.

For more information, see Web authentication settings.

About this task

Use the Atom binding to work securely with services that provide or consume entries described in the Atom Syndication Format and Atom Publishing Protocol.

Procedure

1. Configure the Atom feed service security in an SCA composite definition.

   You can secure services that are exposed over an Atom binding using intents. Administrative and application security must be enabled for the intents to be enforced. The following three intents are valid options for the `requires` attribute on the `binding.atom` element:

   authentication.transport
   Requires any client invoking the service to provide valid authentication information

   confidentiality.transport
   Requires any client invoking the service to do so over a secure transport that provides confidentiality of the transport

   integrity.transport
   Requires any client invoking the service to do so over a secure transport that provides integrity of the transport

   Edit a composite definition that exposes a Java service over the Atom binding so that the exposed service requires a client to authenticate and communicate over a secure transport; for example:

```
<component name="NewsServiceComponent">
  <implementation.java class="test.abdera.NewsServiceImpl"/>
  <service name="NewsService">
    <t:binding.atom uri="/NewsServiceComponent/newsService"
        requires="authentication.transport confidentiality.transport"/>
  </service>
</component>
```

   For information about authorization policy, refer to documentation on SCA authorization and security identity policies.

2. Invoke a secure service that is exposed over an Atom binding.

   You can access the service directly from a browser or a client that supports Atom feeds. To access the feed directly, you can use the uniform resource indicator (URI) that the service specifies.

If the service requires confidentiality or integrity, use the https protocol. If the service requires authentication, the user is prompted by the browser to enter valid credentials. If a Java client is used to access the service, include the authentication information in the HTTP header.

The following example invokes a service using a reference URI. If the service being referenced requires confidentiality or integrity, use the https protocol.

```
<reference name="atomFeed">
    <tuscany:binding.atom uri="https://localhost:9443/newsService"/>
</reference>
```

You can also invoke the service using a reference target:

```
<reference name="atomFeed" target="NewsServiceComponent/NewsService">
    <tuscany:binding.atom/>
</reference>
```

For this example, the invocation is secure only if the service specifies the `confidentiality.transport` or `integrity.transport` intent.

To authenticate when invoking a service over an Atom binding, you have two options:

- Use single sign-on (SSO) to authenticate.

  If SSO is enabled and there has been a successful authentication before, the credentials are propagated with the request. For information about enabling SSO, see the topic on implementing single sign-on to minimize web user authentications.
- Configure an authentication-alias, and send a specific user name and password with the request.

  You can use this option for references in `implementation.java` components. This option is not supported for references in `implementation.widget` components.

  a. Create an authentication-alias using the administrative console Java 2 Connector (J2C) authentication data entry page or wsadmin commands. See topics on J2C authentication data entries.

  b. In the composite definition, define the product SCA namespace and specify the alias name on the `binding.atom` element using the `authentication-alias` attribute.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
...
xmlns:qos="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
...
<reference name="atomFeed" target="NewsServiceComponent/NewsService">
    <tuscany:binding.atom qos:authentication-alias="AtomAlias"/>
</reference>
```

What to do next

Test the service security.

Using Widget implementation in JavaScript with Atom bindings

The JavaScript code in an HTML file can use Service Component Architecture (SCA) references that are defined in a Tuscany Widget implementation. Use Widget implementation to work with data in Atom collections that an SCA service returns in JavaScript.

Before you begin

You can use an Atom binding in an SCA application to expose collections of data as an Atom feed or to reference existing external Atom feeds. If you are unfamiliar with the Atom protocol, refer to documentation on the Atom Syndication Format, an XML-based document format that describes web feeds, and the Atom Publishing Protocol, a protocol for publishing and updating web resources.

About this task

An SCA component can define SCA references for use in JavaScript code. Use Tuscany Widget implementation to define the references. The implementation supports references that use an Atom binding, and does not support the definition of SCA services.

The SCA composite that uses the Widget implementation must be deployed inside a Web application archive (WAR) file.

Procedure

1. Configure a Widget implementation in an SCA composite definition.

   Create an SCA composite definition file for a component that uses Tuscany `implementation.widget`. For example:

   ```
   <composite>
     <component name="Store">
       <tuscany:implementation.widget location="ui/store.html"/>
          <reference name="shoppingCart">
           <tuscany:binding.atom uri="/ShoppingCart/Cart"/>
          </reference>
     </component>
   </composite>
   ```

   This example defines a Store component that uses Tuscany `implementation.widget` in an HTML file at `ui/store.html`.

2. Enable the SCA reference in an HTML file.

   In the HTML file, `ui/store.html`, define two required script elements that enable SCA references. Specify the following for the first element:

   ```
   <script type="text/javascript>>
        dojo.registerModulePath("tuscany", "/Store/tuscany");
        dojo.require("tuscany.AtomService");
   </script>
   ```

   This definition is required when using Atom binding references. The `dojo.registerModulePath` method tells the `dojo` object where to find requirements in the Tuscany namespace. The first argument is always `"tuscany"`. The second argument is specified in the format */SCA_component_name*`/tuscany`. The `dojo.require` statement for `"tuscany.AtomService"` causes the browser to load the JavaScript file from the `/Store/tuscany/AtomService.js` file. The product dynamically generates this file to connect the Atom binding references to Atom services.

   Specify the following for the second element:

   ```
   <script type="text/javascript" src="/Store/store.js"></script>
   ```

   This definition is required in any HTML file that is used as an implementation for a Widget implementation component. For the script `src` attribute, specify the uniform resource identifier (URI) in the format */SCA_component_name/modified_implementation.widget_location_attribute*; for example, `/Store/store.js`. The modified location attribute is the Widget implementation location attribute without a leading path and with a file extension of `.js`.

3. Define the SCA reference in JavaScript in the HTML file.

   An HTML file that contains the above example reference might resemble:

   ```
   //@Reference
   var catalog = new tuscany.sca.Reference("shoppingCart");
   ```

   The `//@Reference` comment is required. The SCA run time interprets the comment in the same manner that a Java class interprets an `@Reference` tag.

4. Use JavaScript to manipulate the feed reference.

   For example, to retrieve an entire feed, the HTML file might use the following example code:

   ```
   var items = shoppingCart.get("");
   ```

   To retrieve a single entry, the implementation might call:

   ```
   var item = shoppingCart.get("Item1");
   ```

   To add a new entry to the feed, the HTML file might use the following example code:

```
var entry =
  '<entry xmlns="http://www.w3.org/2005/Atom"><title>item</title><content type="text/xml">'
   + '<Item xmlns="http://services/">' + <name xmlns="">' + itemName + '</name>'
   + '<price xmlns="">' + itemPrice + '</price>' + '</Item>' + '</content></entry>';

shoppingCart.post(entry);
```

What to do next

Deploy your SCA component in an application.

For additional examples, see the topic on using Widget implementation in JavaScript with Atom or HTTP bindings.

Using HTTP bindings in SCA applications

You can use an HTTP binding with a wire format of JSON-RPC in a Service Component Architecture (SCA) application to expose services to remote web browser clients. JSON-RPC is a remote procedure call (RPC) protocol encoded in the JavaScript Object Notation (JSON) format.

About this task

Use the HTTP binding to expose SCA services for consumption by remote web clients. This topic describes how to expose a Java implementation as a service to be consumed by a browser client using native Dojo interfaces and RPC libraries.

To use native SCA references in JavaScript code, see the topic on using Widget implementation in JavaScript with HTTP bindings.

Procedure

1. Configure the Java service in an SCA composite definition.

   Expose a Java service over the HTTP binding in the composite definition; for example:

```
<composite>
  <service name="EchoService" promote="EchoComponent">
    <interface.java interface="echo.Echo"/>
    <tuscany:binding.http uri="/EchoService"/>
      <tuscany:wireFormat.jsonrpc/>
    </tuscany:binding.http>
  </service>

  <component name="EchoComponent">
    <implementation.java class="echo.EchoComponentImpl"/>
  </component>
</composite>
```

   This example exposes the methods defined in the `echo.Echo` interface to web browser clients. The `echo.EchoComponentImpl` class implements the Echo interface and provides the implementation for the component. The service is exposed on the `"/EchoService"` relative uniform resource identifier (URI).

   The example HTTP binding URI, `/EchoService`, is a relative URI. To run applications that use an HTTP binding in product clusters, specify a relative URI. You cannot run applications in product clusters if the binding specifies an absolute URI, such as `http://localhost:9080/newsService`.

2. Access the service from a web browser.

   For example, to access the `EchoService` service, use the Dojo toolkit application programming interfaces in an HTML file or JavaServer Pages (JSP) file to access the service available at `/EchoService`:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Echo...</title>
<script type="text/javascript" src="./dojo/dojo.js"></script>
<script>
  dojo.require("dojo.io.script")
  dojo.require("dojo.rpc.RpcService")
  dojo.require("dojo.rpc.JsonService")
```

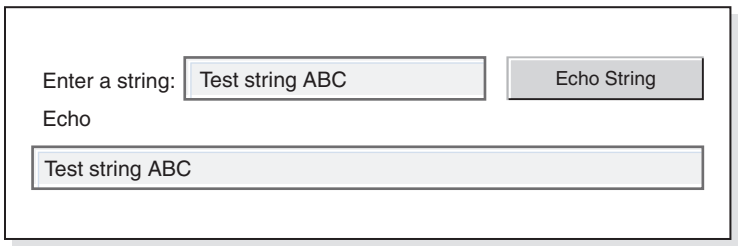
```
  dojo.require("dojo.rpc.JsonpService")

  function filladdr(){
    var nameElement=document.getElementById("name");
    var name=nameElement.value;
    var echo = new dojo.rpc.JsonService("/EchoService?smd");
    echo.echo(name).addCallback(fillmsg);
  }

  function fillmsg(result){
    var line=document.getElementById("line1");
    line.value=result;
  }
</script>
</head>
<body>
Enter a string: <input id="name" type="text">
<input type="submit" value="Echo String" onClick="filladdr()"><br>
<p>Echo </p><input id="line1" type="text" size="50"><br><br>
</body>
</html>
```

The URI passed into the dojo.rpc.JsonService constructor "/EchoService?smd" contains the query string "smd" on the end of the URI specified in the <binding.http> definition in the composite. The query string "smd" on the end of the URI is required when using Dojo clients.

Results

The HTML output for the example output shows two text boxes and a "submit" button. When a user enters text in the first box and clicks the submit button, the following steps take place:

1. The JavaScript code in the filladdr() method obtains the value that was entered in the first text box.
2. The filladdr() method instantiates a dojo.rpc.JsonService object pointed to the URI "/EchoService?smd".
3. The JavaScript code runs the "echo(String)" method on the JsonService object, causing a JSON-RPC request to be sent to "/EchoService?smd".
4. The SCA run time handles the URI request by running the EchoComponentImpl.echo(String) method. The result is returned to the client as an HTTP response.
5. The web client runs the designated callback method, "fillmsg(result)", with the value returned from the service.
6. The JavaScript code in the fillmsg(result) method updates the second text box to contain the text returned from the service invocation.



What to do next

Consider the supported data types for the JSON conversion. Parameters for JSON-RPC requests are sent to the server in JSON format, and must be transformed by the SCA run time for use in the Java implementation. The response to the client is also in JSON format, so the SCA run time converts the value returned from the EchoComponentImpl.echo() method back into JSON. For example, echo("Testing...") might submit the following data to the server:

```
{"params":["Testing..."],"method":"echo","id":1}
```

The Java method `EchoComponentImpl.echo(String message)` is invoked with the String parameter `"Testing..."` and returns the String object `"echo: Testing..."`. The JSON response returned to the web client might look like:

```
{"id":1,"result":"echo: Testing..."}
```

Table 90. Supported data types for JSON conversions. Conversion enables the Web client and Java implementation to use the data.

Data type				
Primitive Type	<==>	JSON	<==>	Primitive Type
Array of Primitive Type	<==>	JSON	<==>	Array of Primitive Type
Java bean	<==>	JSON	<==>	Java bean
List	<==>	JSON	<==>	List
Map	<==>	JSON	<==>	Map
Set	<==>	JSON	<==>	Set

Securing services exposed by HTTP bindings

You can use an HTTP binding with a wire format of JSON-RPC securely in a Service Component Architecture (SCA) application. Use intents to secure services that are exposed by the HTTP binding to remote web browser clients.

Before you begin

JSON-RPC is a remote procedure call (RPC) protocol encoded in the JavaScript Object Notation (JSON) format.

For information about using HTTP bindings in this product, refer to “Using HTTP bindings in SCA applications” on page 812.

Enable administrative and application security, if not done so already. To enforce intents to secure services that are exposed by an HTTP binding, administrative and application security must be enabled.

transition: In Version 8.0, the default value for the web authentication property `webAuthReq` is `persisting`, which enables credential persistence that allows login information to be available to unprotected web clients and enables additional access to user information. You must set the `webAuthReq` property to `lazy` to prevent unprotected web clients additional access to user information through persisting credentials. For more information, see Security hardening features enablement and migration. You can set `webAuthReq` to lazy on the administrative console Web security - General settings page:

1. Click **Security** > **Global security** > **Authentication** > **web and SIP security** > **General settings**.
2. Select **Authenticate only when the URI is protected**, which enables lazy authentication.
3. Click **Apply**.

For more information, see Web authentication settings.

About this task

Use the HTTP binding to secure services. To secure services, configure the `requires` attributes on the `binding.http` element.

Procedure

1. Configure security for the HTTP binding service in an SCA composite definition.

   You can secure services that are exposed over an HTTP binding using intents. The following three intents are valid options for the `requires` attribute on the `binding.http` element:

authentication.transport
Requires any client invoking the service to provide valid authentication information.

confidentiality.transport
Requires any client invoking the service to do so over a secure transport that provides confidentiality of the transport.

integrity.transport
Requires any client invoking the service to do so over a secure transport that provides integrity of the transport.

Edit a composite definition that exposes a Java service over the HTTP binding so that the exposed service requires a client to authenticate and communicate over a secure transport; for example:

```
<composite>
  <service name="EchoService" promote="EchoComponent">
    <interface.java interface="echo.Echo"/>
    <tuscany:binding.http uri="/EchoService" requires="authentication.transport confidentiality.transport"/>
    <tuscany:wireFormat.jsonrpc/>
    </tuscany:binding.http>
  </service>

  <component name="EchoComponent">
    <implementation.java class="echo.EchoComponentImpl"/>
  </component>
</composite>
```

For information about authorization policy, refer to topics on SCA authorization and security identity policies.

2. Invoke a secure service that is exposed over an HTTP binding from a web browser.

   You can access the service directly from a JavaServer Pages (JSP) file or HTML page using the Dojo toolkit application programming interfaces from a web browser.

   If the service requires confidentiality or integrity, use the HTTPS protocol. If the service requires authentication, configure the client application to prompt the user for valid user name and password. If the default product settings enable single sign-on (SSO) and the user has authenticated previously, these credentials are automatically propagated in the request to the service.

What to do next

Test the service security.

Using Widget implementation in JavaScript with HTTP bindings

The JavaScript code in an HTML file can use Service Component Architecture (SCA) references that are defined in a Tuscany Widget implementation. Use Widget implementation to work with data in JavaScript Object Notation (JSON) format that an SCA service returns in JavaScript.

Before you begin

You can use the HTTP binding with a wire format of JSON-RPC to expose SCA services for consumption by remote web browser clients. JSON-RPC is a remote procedure call (RPC) protocol encoded in the JSON format.

About this task

An SCA component can define SCA references for use in JavaScript code. Use Tuscany Widget implementation to define the references. The implementation supports references that use an HTTP binding with a wire format of JSON-RPC, and does not support the definition of SCA services.

The SCA composite that uses the Widget implementation must be deployed inside a web application archive (WAR) file.

Procedure

1. Configure a Widget implementation in an SCA composite definition.

Create an SCA composite definition file for a component that uses Tuscany `implementation.widget`. For example:

```
<composite>
  <component name="Store">
    <tuscany:implementation.widget location="ui/store.html"/>
      <reference name="catalog">
        <tuscany:binding.http uri="/Catalog"/>
          <tuscany:wireFormat.jsonrpc/>
        </tuscany:binding.http>
      </reference>
  </component>
</composite>
```

This example defines a Store component that uses Tuscany `implementation.widget` in an HTML file at `ui/store.html`.

2. Create the HTML file specified in the SCA composite definition for the Widget implementation.

   In the HTML file, define required script elements such as the following:

```
<script type="text/javascript" src="/Store/store.js"></script>
```

   The script `src` attribute points to a JavaScript file that the product dynamically generates to connect the SCA references to their associated services. Specify the uniform resource identifier (URI) in the format */SCA_component_name/modified_implementation.widget_location_attribute*; for example, `/Store/store.js`. The modified location attribute is the location attribute without a leading path and with a file extension of `.js`.

3. Define the SCA reference in JavaScript in the HTML file.

   In the HTML file, define an SCA reference. For example:

```
//@Reference
var catalog = new tuscany.sca.Reference("catalog");
```

4. Add JavaScript code that uses the reference to the HTML file.

   The code used for this example resembles:

```
<script>

  function init() {
    catalog.get().addCallback(catalog_getResponse);
  }

  function catalog_getResponse(items,exception) {
    if(exception){
      alert(exception.message);
      return;
    }
    var catalog = "";
    for (var i=0; i < items.length; i++) {
      var item = items[i].name + ' - ' + items[i].price;
      catalog += '<input name="items" type="checkbox" value="' +
        item + '">' + item + '<br>';
    }
    document.getElementById('catalog').innerHTML=catalog;
  }

</script>
```

   In this example code, the `init` method calls the `get` method on the `catalog` reference. The result is sent to the callback method `callback_getResponse()`. The callback method adds check box elements to the HTML for each item returned from the catalog `get` method.

5. Add the user interface to the HTML file, as needed.

   The `ui/store.html` file used for this example might use the following user interface:

```
<html>
<body onload="init()">
<h1>Store</h1>
   <h2>Catalog</h2>
   <form name="catalogForm">
      <div id="catalog"></div>
      <br>
      <input type="button" onClick="addToCart()" value="Add to Cart">
   </form>
</body>
</html>
```

What to do next

Deploy your SCA component in an application.

When using the Widget implementation, HTTP binding references must be deployed on the same server or cluster as the HTTP binding services that they reference. This limitation is a result of browser limitations on cross-domain JavaScript invocation. If your application defines the reference and service in separate servers or clusters, use a proxy server so that the Widget implementation resource that contains the reference and the HTTP binding service are both accessed using the same HTTP domain.

To resolve HTTP binding references, either use a target attribute, @target, on the reference or specify the URI attribute on the `binding.http` element. For more information, see the topic on resolving SCA references.

For more examples, see the topic on using Widget implementation in JavaScript with Atom or HTTP bindings.

Using Widget implementation in JavaScript with Atom or HTTP bindings

You can use Tuscany Widget implementation to define Service Component Architecture (SCA) references for use in JavaScript code. Use Widget implementation to work with data that an SCA service returns in JavaScript. The data can be in Atom collections or in JavaScript Object Notation (JSON) format.

Before you begin

You can use an Atom binding in an SCA application to expose collections of data as an Atom feed or to reference existing external Atom feeds. If you are unfamiliar with the Atom protocol, refer to documentation on the Atom Syndication Format, an XML-based document format that describes web feeds, and the Atom Publishing Protocol, a protocol for publishing and updating web resources.

You can use the HTTP binding with a wire format of JSON-RPC to expose SCA services for consumption by remote web browser clients. JSON-RPC is a remote procedure call (RPC) protocol encoded in the JSON format.

About this task

An SCA component can define SCA references for use in JavaScript code. Use Tuscany Widget implementation to define the references. The implementation supports references that use an Atom binding or an HTTP binding with a wire format of JSON-RPC, and does not support the definition of SCA services. The implementation also supports the definition of SCA properties.

The SCA composite that uses the Widget implementation must be deployed inside a web application archive (WAR) file.

Procedure

1. Configure a Widget implementation in an SCA composite definition.

   Create an SCA composite definition file for a component that uses Tuscany `implementation.widget`. For example:

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
           targetNamespace="http://store" name="storeWidget">
  <component name="Store">
    <tuscany:implementation.widget location="ui/store.html"/>
 </component>
</composite>
```

This example defines a Store component that uses Tuscany `implementation.widget` in an HTML file at `ui/store.html`. The `implementation.widget` definition is located in the `"http://tuscany.apache.org/xmlns/sca/1.0"` namespace, rather than the `"http://www.osoa.org/xmlns/sca/1.0"` namespace that most SCA artifacts use.

2. Add SCA reference definitions to the SCA composite definition.

   Define one or more references that use an Atom binding or an HTTP binding. An SCA reference that uses an Atom binding resembles:

   ```
   <reference name="shoppingCart">
       <tuscany:binding.atom uri="/ShoppingCart/Cart"/>
   </reference>
   ```

   An SCA reference that uses the HTTP binding with a wire format of JSON-RPC resembles:

   ```
   <reference name="catalog">
       <tuscany:binding.http uri="/Catalog"/>
         <tuscany:wireFormat.jsonrpc/>
       </tuscany:binding.http>
   </reference>
   ```

   For more information on the Atom binding references, see the topics on using Atom bindings. For more information on HTTP binding references, see the topics on using HTTP bindings.

3. Add SCA property definitions to the SCA composite definition.

   You can define one or more properties in the SCA composite definition file. For example:

   ```
   <property name="currency">USD</property>
   ```

4. Add the SCA composite definition to the WAR file.

   Save the SCA composite definition in the `META-INF/sca-deployables/default.composite` file in the WAR file.

   For the example in this topic, the complete composite file is as follows:

   ```
   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
              xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
              targetNamespace="http://store" name="storeWidget">
     <component name="Store">
       <tuscany:implementation.widget location="ui/store.html"/>

         <reference name="shoppingCart">
             <tuscany:binding.atom uri="/ShoppingCart/Cart"/>
         </reference>
         <reference name="shoppingTotal">
             <tuscany:binding.http uri="/ShoppingCart/Total">
               <tuscany:wireFormat.jsonrpc/>
             </tuscany:binding.http>
         </reference>
         <reference name="catalog">
             <tuscany:binding.http uri="/Catalog"/>
               <tuscany:wireFormat.jsonrpc/>
             </tuscany:binding.http>
         </reference>

         <property name="currency">USD</property>

     </component>
   </composite>
   ```

5. Create the HTML file specified in the SCA composite definition for the Widget implementation and add it to the WAR file.

   In the HTML file, define the SCA references and properties and define required script elements as described in topics on using Atom bindings or HTTP bindings. The `ui/store.html` file used for this example resembles:

   ```
   <html>
   <head>
   <title>Store</title>
   ```

```
<script type="text/JavaScript" src="../dojo/dojo.js"></script>
<script type="text/javascript>>
     dojo.registerModulePath("tuscany", "/Store/tuscany");
     dojo.require("tuscany.AtomService");
     dojo.require("dojo.rpc.JsonService");
</script>
<script type="text/JavaScript" src="/Store/store.js"></script>
<script language="JavaScript">

  //@Reference
  var catalog = new tuscany.sca.Reference("catalog");

  //@Reference
  var shoppingCart = new tuscany.sca.Reference("shoppingCart");

  //@Property
  var currency = new tuscany.sca.Property("currency");

  var catalogItems;

  function catalog_getResponse(items,exception) {
    var catalog = "";
    for (var i=0; i < items.length; i++) {
      var item = items[i].name + ' - ' + items[i].price;
      catalog += '<input name="items" type="checkbox" value="' +
        item + '">' + item + '<br>';
    }
    document.getElementById('catalog').innerHTML=catalog;
    catalogItems = items;
  }

  function shoppingCart_getResponse(feed) {
    if (feed != null) {
       var entries = feed.getElementsByTagName("entry");
       var list = "";
       for (var i=0; i < entries.length; i++) {
         var content =
           entries[i].getElementsByTagName("content")[0];
         var name =
           content.getElementsByTagName("name")[0].firstChild.nodeValue;
         var price =
           content.getElementsByTagName("price")[0].firstChild.nodeValue;
         list += name + ' - ' + price + '<br>';
       }
     }
     document.getElementById("shoppingCart").innerHTML = list;
   }

   function shoppingCart_postResponse(entry) {
     shoppingCart.get("").addCallback(shoppingCart_getResponse);
   }

   function addToCart() {
     var items  = document.catalogForm.items;
     var j = 0;
     for (var i=0; i < items.length; i++) {
       if (items[i].checked) {
         var entry =
           '<entry xmlns="http://www.w3.org/2005/Atom"><title>item<content
           type="text/xml">' + '<Item xmlns="http://services/">' + '<name xmlns="">'
           + catalogItems[i].name + '</name>' + '<price xmlns="">' +
           catalogItems[i].price + '</price>' + '</item>' + '</content></entry>';

         shoppingCart.post(entry).addCallback(shoppingCart_postResponse);
         items[i].checked = false;
       }
     }
```

```
    }

    function init() {
      catalog.get().addCallback(catalog_getResponse);
      shoppingCart.get("").addCallback(shoppingCart_getResponse);
    }
</script>
</head>

<body onload="init()">
<h1>Store</h1>
 <div id="store">
   <h2>Catalog</h2>
   <form name="catalogForm">
      <div id="catalog"></div>
      <br>
      <input type="button" onClick="addToCart()" value="Add to Cart">
   </form>
   <br>
   <h2>Your Shopping Cart</h2>
   <form name="shoppingCartForm">
      <div id="shoppingCart"></div>
      <br>
      <div id="total"></div>
      <br>
   </form>
 </div>
</body>
</html>
```

What to do next

Deploy your SCA component in an application.

Resolving SCA references

During application assembly or deployment, a reference (a service dependency) is typically resolved to an actual deployed SCA service.

About this task

You can specify the target endpoint of a Service Component Architecture (SCA) reference in any of the following ways:

- Using the @target attribute on the reference element in order to target a reference to a component service within the domain
- For OASIS composites, using the @target attribute of the `binding.ws` element to target a web service reference to an SCA web service within the domain
- Using the @uri attribute of the binding element to specify a binding-specific endpoint

Using the @target attribute on the reference element

Use this option when the target service is another SCA service that is in the same domain as the client component, or rather, the component with the reference.

This @target attribute is supported for `binding.sca` for both OSOA and OASIS composites. In an OSOA composite, you can optionally include a `binding.sca` element within a reference element that uses the @target attribute. In an OASIS composite, you cannot include a `binding.sca` element within a reference element that uses the @target attribute.

The @target attribute is supported for the following bindings for OSOA composites only:
- binding.ws
- binding.atom

- binding.http
- binding.json

When a reference uses the @target attribute, the client does not need to know the endpoint address of a service. It is determined during run time. Also, the @target attribute does not need to be updated when the target service is deployed to a new server with a different address.

If you use this approach, remember that you must use bindings of the same type, meaning that the reference must share a common binding with the service it is targeting.

Using the @target attribute on the binding.ws element

Use this option to wire an SCA web service reference to an SCA web service within the same domain.

This option is supported for OASIS composites only.

Using a binding-specific endpoint

You must resolve an SCA reference using a binding-specific endpoint if you invoke non-SCA services over non-default bindings or if you have compatible SCA services that are hosted in another domain.

In general, obtain the binding-specific endpoint from the service provider.

If your target service is another SCA service, see the documentation for configuring the particular SCA binding to learn more about which binding-specific endpoint is used for a given service deployment over a particular binding.

Procedure

1. Determine from the service provider whether the service that you are consuming is an SCA service within the same domain as your client.
2. Determine the binding that your client uses to consume this service.

   If the target service is an SCA service, the binding that you use is based on the bindings over which the service is exposed. If the service is not an SCA service, the binding depends on the technology over which the service is provided. For example, services offered over SOAP/HTTP use the SCA web services binding.
3. If the SCA service is hosted in the same domain as your client, use the @target attribute to resolve a reference to a component service within the domain.

   The following examples demonstrate using the @target attribute. The syntax for the `<reference>` element is the same for the different SCA binding types.

   - SCA default binding

     Target component

     ```
     <component name="TargetComponent">
       <service name="BankService"/>
     </component>
     ```

     Client component

     ```
     <component name="ClientComponent">
       <reference name="myReference" target="TargetComponent"/>
     </component>
     ```

   - SCA web service binding

     Target component

     ```
     <component name="TargetComponent">
       <service name="BankService">
         <interface.wsdl ....>
         <binding.ws/>
       </service>
     </component>
     ```

     Client component (OSOA example)

```
<component name="ClientComponent">
  <reference name="myReference" target="TargetComponent">
    <interface.wsdl ....>
    <binding.ws/>  <!-- The client does not have to specify endpoint details. -->
  </reference>
</component>
```

Client component (OASIS example)

```
<component name="ClientComponent">
  <reference name="myReference">
    <interface.wsdl ....>
    <binding.ws target="TargetComponent"/>
  </reference>
</component>
```

- SCA Atom binding

  Target component

  ```
  <component name="NewsServiceComponent">
    <service name="NewsService">
      <t:binding.atom uri="/newsService"/>
    </service>
  </component>
  ```

  Client component

  ```
  <component name="NewsComponent">
    <reference name="newsServiceRef" target="NewsServiceComponent/NewsService">
      <t:binding.atom/> <!-- The client does not need to specify endpoint details -->
    </reference>
  </component>
  ```

- SCA HTTP binding

  Target component

  ```
  <component name="Catalog">
    <service name="Catalog">
      <t:binding.http>
        <t:wireFormat.jsonrpc/>
      </t:binding.http>
    </service>
  </component>
  ```

  Client component

  ```
  <component name="Store">
     <t:implementation.widget location="store.html"/>
     <reference name="catalog" target="Catalog/Catalog">
        <t:binding.http/> <!-- The client does not need to specify endpoint details -->
           <t:wireFormat.jsonrpc/>
        </t:binding.http>
     </reference>
  </component>
  ```

4. Resolve the SCA reference by using a binding-specific endpoint if you are invoking non-SCA services over non-default binding or if you have compatible SCA services that are hosted in another domain.

   The following examples demonstrate using the binding-specific endpoint for the client component:

   - SCA web service client component

     ```
     <component name="ClientComponent">
       <reference name="myReference">
           <!-- The exact URL is obtained from a service provider. -->
         <binding.ws uri="http://www.mybank.com:9080/MyBank/AccountService/services>
       </reference>
     </component>
     ```

   - SCA `binding.atom` client component

     ```
     <component name="Aggregator">
       <reference name="atomFeed1">
         <t:binding.atom
           uri="http://www.ibm.com/developerworks/views/webservices/rss/libraryview.jsp"/>
       </reference>
     </component>
     ```

   - SCA `binding.http` client component

```
<component name="Store">
  <reference name="catalog">
    <t:binding.http uri="/catalog">
      <t:wireFormat.jsonrpc/>
    </t:binding.http>
  </reference>
</component>
```

See the documentation for configuring the particular SCA binding to learn more about binding-specific endpoint resolution for these SCA binding types.

Results

You have identified your SCA client's reference to a target service that it will consume.

Routing HTTP requests to an SCA service when using an external web server

If you are using an external web server to route requests to an SCA service that is exposed over the SCA web services, Atom or HTTP binding, you must define the endpoints of the SCA service to the Web Server HTTP plug-in.

About this task

Requests to services that are exposed over the SCA binding that use the proxy server type that is provided with WebSphere Application Server are routed over the specified proxy by default.

However, if your configuration uses an external web server with the HTTP plug-in for WebSphere Application Server and you want requests to services that are exposed over the SCA binding to route through the external web server, you must define the endpoints for the SCA service by adding the service URL patterns to the `plugin-cfg.xml` file for the Application Server.

Procedure

1. Obtain the URL patterns for each service.

   You can obtain the URL patterns in one of the following ways:

   - Use the message, which is located in the server log file, that indicates the web application is successfully created.

     During service startup, for each service that is exposed over an SCA binding, a dynamic web application is created and configured with the URI of the service. This process is described as an informational message within the server log file as shown in the following example.

     In the following example message, the helloworldws composition unit contains the AsyncTranslatorService service, which is exposed over the SCA web service binding. This message provides the necessary context root and URL pattern for the service, which you must add to the `plugin-cfg.xml` file.

```
/AsynchTranslatorComponent/AsynchTranslatorService/*

[[11/18/08 10:10:52:156 EST] 00000070 servlet I com.ibm.ws.webcontainer.servlet.ServletWrapper init
SRVE0242I: [helloworldws]
[/AsynchTranslatorComponent/AsynchTranslatorService]
[SCA_WS_BINDING_IMPL_CLASS_PLACEHOLDER]: Initialization successful.
[11/18/08 10:10:52:156 EST] 00000070 WASAxis2Exten I
WSWS7037I: The /* URL pattern was configured for the SCA_WS_BINDING_IMPL_CLASS_PLACEHOLDER servlet located
in the SCAWSBindSERV_AsynchTranslatorComponent_AsynchTranslatorService.war web module.
```

   - Use the `warinfo.props` WebSphere configuration repository file.

     For each composition unit that has at least one service or reference exposed over the binding, a single `warinfo.props` file is generated during deployment. This file contains configuration information for each dynamic web application that starts during the server startup process.

     The `warinfo.props` file is located in, for example, the *profile_root*`\SOAAppSrv01\config\cells\cell1\cus\helloworldws\cver\BASE\meta\warinfo.props` directory.

The file contains an entry for each dynamic web application. For example:

```
#
#Tue Nov 18 10:10:37 EST 2008
SCAWSBindSERV_AsynchTranslatorComponent_
AsynchTranslatorService.war=AsynchTranslatorComponent/
AsynchTranslatorService\:default_host\:false\:false\:false
```

The value immediately following the `war=` and ending prior to the `\:` is the context root for the web application. In this example, the context root is **AsynchTranslatorComponent/ AsynchTranslatorService**.

2. Add the values for each dynamic web application to the `plugin-cfg.xml` file.

   After obtaining all of the entries from each of the services that you want to define to the proxy server, add the values to the `plugin-cfg.xml` file. It is important that you add the URI to the specific UriGroup that contains a server and hosts the proxied service because multiple UriGroups might exist. If this process is not done correctly, an HTTP 404 message results.

   In the following example, see the **AsynchTranslatorComponent/AsynchTranslatorService** entry that has been added to the list of URI patterns:

```
<UriGroup Name="default_host_Cluster2_URIs">
? <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/IBM_WS_SYS_RESPONSESERVLET/*" />
?  <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/IBM_WS_SYS_RESPONSESERVLET/*.jsp" />
?  <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/IBM_WS_SYS_RESPONSESERVLET/*.jsv" />
?  <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/IBM_WS_SYS_RESPONSESERVLET/*.jsw" />
?  <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/IBM_WS_SYS_RESPONSESERVLET/j_security_check" />
?  <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/IBM_WS_SYS_RESPONSESERVLET/ibm_security_logout" />

?  <Uri AffinityCookie="JSESSIONID"
AffinityURLIdentifier="jsessionid"
Name="/AsynchTranslatorComponent/AsynchTranslatorService/*" /> </UriGroup>
```

Results

You have configured the endpoints for SCA services to route requests through an external web server configured with the HTTP plug-in for WebSphere Application Server

Interoperability between Open SCA client services and WebSphere Process Server SCA modules

Support for Service Component Architecture (SCA) provides a simple, yet powerful programming model for constructing applications based on the Open SCA specifications. The SCA modules of WebSphere Process Server use import and export bindings to interoperate with Open SCA services.

An Open SCA application invokes WebSphere Process Server SCA applications using an export binding. An Open SCA application receives a call from a WebSphere Process Server SCA application using an import binding.

When building an Open SCA service client which will invoke WebSphere Process Server SCA module services, start with an existing WSDL file for all supported bindings except Enterprise JavaBeans (EJB) bindings.

Generate Java interfaces from the Web Services Description Language (WSDL) file. Do not port modules from WebSphere Process Server SCA to Open SCA and do not port modules from Open SCA to WebSphere Process Server SCA.

In general, Java artifacts are not reusable across WebSphere Process Server SCA modules and Open SCA applications, even when one artifact invokes the other over a common WSDL interface. You must use appropriate tooling to separately generate, from the WSDL interface, the Java interfaces and classes that are used in the Open SCA applications and the Java interfaces and classes that are used in the WebSphere Process Server SCA modules. For Open SCA applications, use the **wsimport** command-line tool or Rational Application Developer with SCA function. For WebSphere Process Server SCA modules, use a tool such as WebSphere Integration Developer.

Bindings that support interoperability between Open SCA client services and WebSphere Process Server SCA modules

Currently, four bindings support interoperability between Open SCA client services and WebSphere Process Server SCA modules:

- SCA binding
- EJB binding
- Web service binding
- JMS binding

SCA binding

When configuring a reference to a WebSphere Process Server SCA module export over SCA binding the following restrictions apply:

- Only synchronous invocations of request-response (two-way) operations are supported.
- All components or modules must be in the same WebSphere Application Server cell definition. The SCA binding is not supported across the cell.
- Both the Open SCA component and the WebSphere Process Server SCA module must use a WSDL interface that meets the following requirements:
  - Web Services Description Language (WSDL) Version 1.1
  - WS-I Basic Profile Version 1.1
  - Simple SOAP Binding Profile 1.0 standards
  - Document literal style

Constructing a URI for a WebSphere Process Server SCA module export over SCA binding

WebSphere Process Server SCA module export is exposed using a uniform resource indicator (URI) that is constructed as follows:

module_name/export_name

The *module_name* and *export_name* variables are defined in the WebSphere Process Server application. You can view the values from either WebSphere Integration Developer or from the administrative console.

Specify this URI as the reference target URI or as the URI in the `binding.sca` element in the `.composite` file. For example, for a WebSphere Process Server module named `HelloWorldModule` and a WebSphere Process Server export named `/test/sca/ClassicHelloWorld`, the `.composite` file resembles the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" autowire="false"
       name="HelloWorldClientComposite" targetNamespace="http://sca.test">
  <component name="HelloWorldClientComponent">
    <implementation.java class="test.sca.open.OpenHelloWorldClient"/>
    <reference name="classicHWReference" target="HelloWorldModule/test/sca/ClassicHelloWorld">
      <interface.wsdl interface="http://sca.test#wsdl.interface(HelloWorldInterface)"/>
      <binding.sca/>
    </reference>
  </component>
</composite>
```

or:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" autowire="false"
       name="HelloWorldClientComposite" targetNamespace="http://sca.test">
  <component name="HelloWorldClientComponent">
```

```
    <implementation.java class="test.sca.open.OpenHelloWorldClient"/>
    <reference name="classicHWReference">
      <interface.wsdl interface="http://sca.test#wsdl.interface(HelloWorldInterface)"/>
      <binding.sca uri="HelloWorldModule/test/sca/ClassicHelloWorld"//>
    </reference>
  </component>
</composite>
```

EJB binding

When configuring a reference to a WebSphere Process Server SCA module export over an EJB binding, the following restrictions apply:

- Only synchronous invocations are supported.
- Only EJB 3.x protocol is supported.
- Both the Open SCA component and WebSphere Process Server SCA module must use a Java interface that conforms to the EJB 3.x programming model.

Web service (JAX-WS) binding

When configuring a reference to a WebSphere Process Server SCA module over a Web service binding, the following features are supported:

- Asynchronous (one-way) and synchronous (request or response) invocations
- The SOAP1.1/HTTP or SOAP1.2/HTTP protocols
- Web Services Atomic Transaction and Web Services Security qualities of service

When configuring a reference to a WebSphere Process Server SCA module over a Web service binding, the following restrictions apply:

- Callback is not supported for non-SCA services like WebSphere Process Server SCA export modules.
- Both the Open SCA component and the WebSphere Process Server SCA module must use a WSDL interface that meets the following requirements:
  - Web Services Description Language (WSDL) Version 1.1
  - WS-I Basic Profile Version 1.1
  - Simple SOAP Binding Profile 1.0 standards
  - Document literal style

JMS binding

When configuring a reference to a WebSphere Process Server SCA module over a JMS binding, the following features are supported:

- Asynchronous (one-way) and synchronous (request or response) invocations
- JMS provider platform messaging (JMS binding)
- JMS provider WebSphere MQ (WebSphere MQ JMS binding)

When configuring a reference to a WebSphere Process Server SCA module over a JMS binding, the following restrictions apply:

- When using `commonj.sdo.DataObject` as a parameter type in Java, `wireFormat.jmsObject` is not supported. Although both the SCA and WebSphere Process Server application Java programming models support the use of the same Java-serializable type, `commonj.sdo.DataObject`, and `wireFormat.jmsObject` results in Java serialization, the `commonj.sdo.DataObject` interface is backed by different implementations in each of the two environments. Thus, this wire format is not an interoperable option. Instead, use an XML wire format, such as `wireFormat.jmsTextXML` or `wireFormat.jmsBytesXML`, or use `wireFormat.jmsdefault`.
- Callback is not supported.

Data types

Business graphs are not interoperable across any SCA bindings and, therefore, are not supported in interfaces used to interoperate with SCA.

Creating wire format handlers

You can use a wire format handler to transform data between a Service Component Architecture (SCA) application and the application binding.

Before you begin

This topic assumes that you have an SCA application with a specified binding. The product supports wire format handling for a Java Message Service (JMS) binding.

Unless otherwise specified, the information in this topic pertains to both OSOA and OASIS applications.

About this task

You can implement a wire format handler that converts data between an application wire format and a JMS Message wire format that is used by the JMS destination of a JMS binding.

Creating a wire format handler consists of two main steps:

1. Implement a Java interface that defines the wire format handler.
2. Configure the wire format handler in the composite definition file of the SCA application.

Procedure

1. Implement the wire format handler interface.

   The wire format handler provides function that transforms one form (source) to another. The result of a transformation is mapped into either object source or JMS message source. The wire format handler must implement the WireFormatHandler interface, which has methods that map into the two source types.

   See the following example wire format handler interface:

```
package com.ibm.websphere.soa.sca.wireformat;

public interface WireFormatHandler {

    /**
  * Transform.
  *
  * @param source
  *            the source
  *
  * @return the object
  *
  * @throws WireFormatHandlerException
  *             the wire format handler exception
  */
    public Object transform(Object source)
        throws WireFormatHandlerException;

    /**
  * Sets the wire format context.
  *
  * @param ctx
  *            the new wire format context
  */
    public void setWireFormatContext(WireFormatContext ctx);

    /**
  * Gets the wire format context.
  *
  * @return the wire format context
  */
    public WireFormatContext getWireFormatContext();}
```

In the `public Object transform` method implementation, the wire format handler transforms data from the source object to a target object. If an error occurs during the transformation, the data handler implementation throws a WireFormatHandlerException.

The `public void setWireFormatContext` method implementation sets the runtime context of the wire format handler. The `public WireFormatContext getWireFormatContext` method implementation gets the runtime context of the wire format handler. Even if you do not intend to use the context object, you must implement these methods.

The wire format context contains runtime contextual information passed from the caller to the wire format handler. The WireFormatContext interface specifies the runtime context of the wire format handler. Each wire format handler implementation must implement the `setWireFormatContext` method of the WireFormatContext interface.

The WireFormatContext class provides a java.util.Map interface where you can set property key and value pairs in the context. The WireFormatContext interface also provides several methods to extract useful information about the current context such as component and service names, invocation types, and the ability to mark exceptions. Refer to the Java documentation for the com.ibm.websphere.soa.sca.wireformat.WireFormatContext interface for a complete list of methods.

2. Configure the wire format handler in the SCA composite definition file.

   Configuring a wire format handler consists of specifying the configuration properties at authoring time and then accessing the configuration properties at run time. Update the composite definition file of your SCA composite to instruct the component to use the wire format handler class; for example:

   OSOA

```
xmlns:fep="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"

<component name="Account">
  <implementation.java class="helloworld.AccountComponent"/>
  <service name="Account">
    <binding.jms>
      <destination ...>
      <response>
        ...
      </response>
      <fep:wireFormat.jmsCustom class="helloworld.custom.FirstWFH"/>
    </binding.jms>
  </service>
</component>
```

   OASIS

```
xmlns:fep="http://www.ibm.com/xmlns/prod/websphere/sca/1.1"

<component name="Account">
  <implementation.java class="helloworld.AccountComponent"/>
  <service name="Account">
    <binding.jms>
      <fep:wireFormat.jmsCustom class="helloworld.custom.FirstWFH"/>
      <destination ...>
      <response>
        ...
      </response>
    </binding.jms>
  </service>
</component>
```

What to do next

Deploy your SCA application and test its behavior.

Wire format handler errors

A wire format handler can transform data between a Service Component Architecture (SCA) application and the application binding. This topic discusses the different types of exceptions and errors that can result at run time in an SCA application that uses wire format handlers. The product supports wire format handling for a Java Message Service (JMS) binding.

The information in this topic pertains to both OSOA and OASIS applications.
- Business exceptions
- Runtime exceptions
- Service and reference exceptions
- Exception handling from a wire format handler

Business exceptions

Business exceptions are business errors or exceptions that occur during processing.

A business exception is a checked exception. It inherits from the Exception class. Client code must handle a checked exception in a catch clause or with a throws clause.

When a business exception occurs, a checked exception object results from the service business exception in the SCA application. The object calls the wire format handler that is configured on the service.

Runtime exceptions

Runtime exceptions are exceptions that occur in the SCA application from processing of a request. Runtime exceptions do not correspond to business exceptions. Unlike business exceptions, runtime exceptions are not defined on the interface.

You can propagate runtime exceptions to the client application so that the client application handles the exceptions. For example, if a client sends a request to create a customer to an SCA application and an authorization error occurs during processing of this request, the SCA component throws a runtime exception. This runtime exception must be propagated back to the calling client so it can act on the authorization request. You can use a wire format handler configured on the reference to propagate runtime exceptions back to the calling client application.

Service and reference exceptions

To handle both business exceptions and runtime exceptions, you can set up exception handling on the application bindings. Handling exceptions on either the reference or service on a binding is optional.

You likely create exception wire format handlers for the most common errors that occur at run time in your application:
- When the exception thrown by the SCA application service must be transformed to a value the client can understand.
- When the incoming exception received on a reference must be transformed to a value the client code can understand.

Exception handling from a wire format handler

An exception handler is a wire format handler. Business and runtime exception handling is done from the wire format handler An exception wire format handler transforms specific exceptions into objects that can be understood by the client application.

Both service and reference bindings can use an exception wire format handler. You can configure an exception wire format handler to do the following:
- Handle runtime exceptions that are received from external applications (reference)
- Send runtime exceptions that have occurred in your SCA application to the external applications (service)

On the service implementation of a wire format handler, the implementation can examine the WireFormatContext.INVOCATION value from the context to find that an exception was thrown. The wire format handler might handle normal invocations and exceptions differently. See the following sample code:

```
public Message transformIntoJMSMessage(Session session, Object source)
   throws JMSException, WireFormatHandlerException {

   WireFormatContext.INVOCATION invocationType =
      (WireFormatContext.INVOCATION) context.get(WireFormatContext.INVOCATION_TYPE);

   if (invocationType == WireFormatContext.INVOCATION.TYPE_EXCEPTION) {
      return handleException(session, source);
   } else if (invocationType == WireFormatContext.INVOCATION.TYPE_RESPONSE) {
      return handleResponse(session, source);
   } else {
      throw new WireFormatHandlerException("Unexpected Invocation Type.");
   }
}
```

On the reference implementation of a wire format handler, the implementation must determine at run time if the incoming response is of type fault. The WireFormatContext.INVOCATION value must be initially set to TYPE_RESPONSE. The implementation examines incoming messages. When an incoming message is an exception rather than a normal response, the implementation must set the invocation type to TYPE_EXCEPTION and throw an appropriate exception. See the following code sample:

```
// On Reference-side, this is for an incoming response
public Object transformFromJMSMessage(Message source)
  throws WireFormatHandlerException {

  ObjectMessage msg = null;
  Object payload = null;
  Object[] payloadArray = null;
  Integer payloadInt = null;
  Object payloadItem = null;

  try {
    msg = (ObjectMessage)source;
  } catch (ClassCastException cce) {
      throw new WireFormatHandlerException("Did not receive ObjectMessage", cce);
  }
  try {
    payload = msg.getObject();
  } catch (JMSException e) {
      throw new WireFormatHandlerException("Cannot get Object from message using getObject()", e);
  }

  if (payload instanceof Integer) {
    // Expected type returned
    payloadInt  = (Integer)payload;
    Integer code = new Integer(0);
    Integer one = new Integer(1);
    code = payloadInt + one;
    return code;
  } else {
    // Response is some type of exception...

    // Change the response type to Exception
    context.put(WireFormatContext.INVOCATION_TYPE, WireFormatContext.INVOCATION.TYPE_EXCEPTION);

    if (payload instanceof Throwable) {
      Throwable thw = (Throwable) payload;

      ////////////////////////////////////////////////////
      // TO DO: Map to checked business exception and throw it
      ////////////////////////////////////////////////////

    } else {
      return new ServiceRuntimeException(((Throwable) payload).getMessage());
      // Unchecked
    }
```

```
  } else {
    throw new WireFormatHandlerException("Unexpected Object returned: " + payload);
    // Unexpected or unknown object returned.
    }
  }
}
```

Interoperating between SCA OASIS and OSOA composites

Service Component Architecture (SCA) applications are based on either the Open SOA Collaboration (OSOA) Version 1.0 SCA specifications or the OASIS SCA Version 1.1 specifications. You cannot mix OSOA and OASIS SCA artifacts, such as `.composite` files or `sca-contribution.xml` files, within the same asset. However, you can wire OASIS and OSOA SCA components together when both SCA composites are running in a single product cell.

Before you begin

For information about the differences between OSOA and OASIS specifications and about the differences between OSOA and OASIS in SCA applications, see the SCA overview topic.

About this task

This topic describes how you might wire OSOA and OASIS components together so that the components, although in separate SCA applications, can interoperate while both applications are running in the same product cell. The procedure provides a simple example that shows how to wire an OASIS component reference to an OSOA component service.

When wiring components for interoperation between OSOA and OASIS in SCA applications, consider the following restrictions:

- OSOA and OASIS components must interoperate over `@Remotable` interfaces. Local interfaces cannot be used and result in an exception error.
- Interfaces must be annotated in a way consistent with the run time in which they are deployed.
  - For an OASIS component to reference an OSOA component, the OASIS reference type must contain OASIS annotations, such as `org.oasisopen.sca.annotation.Remotable`.
  - For an OSOA component to reference an OASIS component, the OSOA reference type must contain OSOA annotations, such as `org.osoa.sca.annotations.Remotable`.
- For a reference in an OASIS component to target an OSOA component, the uniform resource identifier (URI) form must be `component` or `component/service`. The `component/service/binding` URI form is not supported.
- OASIS interfaces marked with the `asyncInvocation` intent cannot interoperate with OSOA services.
- The product does not perform interface matching between OASIS and OSOA services. Incompatible interfaces result in a runtime exception.
- The product does not perform policy matching between OASIS and OSOA services. Incompatible policies result in a runtime exception.

Procedure

1. Create an OSOA SCA application.
   a. Define an OSOA composite with a single component for the application.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="InteropOSOARemote">
   <component name="HelloWorldService">
      <implementation.java class="interop.osoa.HelloWorldServiceImpl"/>
      <service name="HelloWorld"/>
   </component>
</composite>
```

   b. Create a service interface that has a Java interface marked as remotable. The `@Remotable` annotation comes from the OSOA annotations package.

```
package interop.osoa;

import org.osoa.sca.annotations.Remotable;

@Remotable
interface HelloWorld {
   public String sayHello(String text);
}
```

c. Create a service implementation.

   For this example, the service implementation adds the string "Hello " to the start of the text that it is sent, and then returns the modified text:

```
package interop.osoa;

public class HelloWorldServiceImpl implements HelloWorld {
   public String sayHello(String text) {
      return "Hello " + text;
   }
}
```

2. Create an OASIS SCA application and wire a component reference to the service that is defined in the OSOA application.

   a. Define an OASIS composite with a single component and with a reference to the service in the OSOA application.

```
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" name="InteropOASIS">
   <component name="HelloWorldClient">
      <implementation.java class="interop.oasis.HelloWorldClientImpl"/>
      <service name="HelloWorld"/>
      <reference name="helloWorldOSOA" target="HelloWorldService"/>
   </component>
</composite>
```

   b. Create a reference interface that has a Java interface marked as remotable.

      The reference interface is compatible with the HelloWorld service interface that is defined in the OSOA application but it is a different Java interface. The reference interface is in a different Java package; the @Remotable annotation comes from the OASIS annotations package.

```
package interop.oasis;

import org.oasisopen.sca.annotation.Remotable;

@Remotable
interface HelloWorld {
   public String sayHello(String text);
}
```

   c. Create a service implementation that implements the HelloWorld interface and uses a reference to the OSOA version of the component.

      For this example, the reference determines the text to return to the caller.

```
package interop.oasis;

import org.oasisopen.sca.annotation.Reference;

public class HelloWorldClientImpl implements HelloWorld {

   @Reference
   protected HelloWorld helloWorldOSOA;

   public String sayHello(String text) {
      return helloWorldOSOA.sayHello(text);
   }
}
```

3. Deploy the SCA applications into the same, or separate, business-level applications in the same cell.

   For OSOA, an sca-contribution.xml file is not required if there is only one default.composite file in the Java archive (JAR). An sca-contribution.xml file can reside in the META-INF/ directory or in a subdirectory.

   For OASIS, an sca-contribution.xml file is required and must reside in the META-INF/ directory, and not in a subdirectory.

What to do next

To test the applications, you can call the HelloWorld service of the OASIS component (the HelloWorldClient component in step 2) and have the service return the string that it retrieves from the OSOA component (the HelloWorldService component in step 1).

Using existing Java EE modules and components as SCA implementations

You can use the Service Component Architecture (SCA) programming model to invoke business services in Java Platform, Enterprise Edition (Java EE) components.

About this task

The SCA programming model supports Java EE integration. You can expose Enterprise JavaBeans (EJB) stateless session beans as SCA services. You can take advantage of SCA annotations to enable the Java EE components, such as session beans, message driven beans, or web components to consume SCA services. You also can rewire EJB references to SCA services. Thus, you can enable an existing Java EE component so that it is recognized as an SCA component and so that the component can participate in an SCA composite.

Procedure

1. Use non-SCA enhanced Java EE applications as SCA component implementations.
2. Use SCA enhanced Java EE applications as SCA component implementations.

   For this step, create an `application.composite` file. Then, determine the Java EE component type that you want to consume SCA services. Depending on the Java EE component type, use SCA annotations:

   - Use SCA annotations with web modules.
   - Use SCA annotations with session beans.
   - Use SCA annotations with message-driven beans.

   If your application uses security, specify security roles and runAs identity in the Java EE application implementation instead of in the composite. Authorization policy is enforced by the implementation.
3. Rewire EJB references to SCA references.
4. Deploy an SCA composite using a Java EE application as a component implementation.

   You can deploy an SCA composite that uses an `implementation.jee` defining a Java EE application as a component implementation. Deploy the SCA composite that uses the application as a component implementation along with the enterprise archive (EAR) file. Add the SCA composite and Java EE application as composition units of a business-level application.

   a. Import the EAR file as an asset.
   b. Import the SCA composite Java archive (JAR) file as an asset.
   c. Create an empty business-level application.
   d. Add the EAR file asset to the business-level application.
   e. Add the SCA composite asset to the business-level application. Map the SCA composite to the same target server as the EAR file.
   f. If you are rewiring EJB references, set the starting weight of the EAR file to a greater value than the starting weight of the SCA composite. The SCA composite then starts before the EAR file.

   When deploying an SCA JAR contribution that has a deployable SCA composite with `implementation.jee` for one or more of its SCA components, ensure that the following requirements are met:

   - Deploy the SCA deployable composite and all the Java EE applications used for the SCA component implementation in the same business-level application.

- Add all the Java EE applications referenced by the `archive` attribute of the `implementation.jee` directive to the business-level application before adding the SCA deployable composite.
- Map all the deployed components of an SCA business-level application to the same target server or cluster.
- When using an SCA enhanced EAR file that requires injection of values for SCA references, properties, context or component names in Java EE modules such as stateless session beans, servlets, or JSP files, the deployed SCA asset must start before the Java EE asset. If necessary, set the weights of the composition units so that the deployed SCA asset starts before the Java EE asset.
- If you update Java EE composition units, deploy the SCA composition unit again to apply the changes in the SCA component that uses Java EE implementation.

5. Start the business-level application.

Results

You now have defined and deployed Java EE components to take advantage of the SCA programming model. The SCA composite and the EAR file are deployed and started, and the Java EE application is able to participate in the SCA domain.

Invoke the EJB services exposed as SCA services using the SCA programming model.

Example

The product provides the HelloJee sample to show how to use SCA annotations within Java EE components so that these components can consume SCA services.

This sample uses the `HelloJeeEar.ear`, `HelloJeeEnhancedEar.ear`, and `HelloJeeSca.jar` SCA sample files. To download HelloJee sample files from a product website:

1. Go to the **Samples, Version 8.5** information center.
2. On the **Downloads** tab, click **FTP** or **HTTP** in the **Service Component Architecture** section.
3. In the authentication window, click **OK**.
4. From the **SCA.zip** compressed file, download prebuilt `HelloJeeEar.ear`, `HelloJeeEnhancedEar.ear`, and `HelloJeeSca.jar` files in the `SCA/installableApps` directory.

   If you want to build your own deployable files, download the `SCA/HelloJee` directory and follow instructions in `SCA/HelloJee/documentation/readme.html` to build the files.

Briefly, to deploy the HelloJee sample, do the following:

1. Import as assets the `HelloJeeEar.ear`, `HelloJeeEnhancedEar.ear`, and `HelloJeeSca.jar` files.
2. Create a business-level application named `HelloJeeBla`.
3. Add the `HelloJeeEar.ear` and `HelloJeeEnhancedEar.ear` assets to the business-level application.
4. Add the `HelloJeeSca.jar` asset to the business-level application. When adding the asset, ensure that it is targeted to the same server or node as the EAR files added in step 3. If you are adding the JAR file using the administrative console, ensure that components HelloJeeEnhancedComponent and HelloJeeComponent are listed on the Set Java EE composition unit relationships page. Also ensure that HelloJeeEnhancedComponent associates with the HelloJeeEnhancedEar composition unit and HelloJeeComponent associates with the HelloJeeEar composition unit.
5. Start the business-level application.
6. Run the sample application. Open a web browser on a URL that accesses a sample. The following URLs use `localhost` for host name and `9080` for port number. Use the host name and `WC_defaulthost` port number that is correct for your installation.
   - Access the non-SCA enhanced EJB as is: `http://localhost:9080/HelloJeeWeb/JeeEjbClient`

- Access the non-SCA enhanced EJB as an SCA service: `http://localhost:9080/HelloJeeWeb/JeeScaClient`
- Access the SCA enhanced EJB as is: `http://localhost:9080/HelloJeeEnhancedWeb/JeeEEjbClient`
- Access the SCA enhanced EJB service or reference as an SCA service or reference and demonstrate SCA annotations: `http://localhost:9080/HelloJeeEnhancedWeb/JeeEScaClient`

Using non-SCA enhanced Java EE applications as SCA component implementations

You can use an existing Java Platform, Enterprise Edition (Java EE) application as a Service Component Architecture (SCA) component without requiring SCA annotations or composite files.

Before you begin

Identify the Java EE application that contains business logic in Enterprise JavaBeans (EJB) or web modules to enable in the SCA environment.

A Java EE application is also called an enterprise application or enterprise archive (EAR) file.

About this task

The SCA programming model supports Java EE integration. You can expose EJB stateless session beans as SCA services by enabling an existing enterprise application module to be recognized as an SCA component and participate in an SCA composite.

You can use an enterprise application as an SCA component implementation, without requiring any SCA annotations or composite files in the application. Such an application is denoted as a non-SCA enhanced enterprise application.

The EJB business interfaces of stateless session beans configured in EJB modules are exposed as SCA services. The EJB business interfaces of EJB references configured in EJB and web modules are exposed as SCA references.

The component type of a non-SCA enhanced enterprise application is defined as follows:

- Each EJB 3.x session bean business interface with unqualified name `intf` of a session bean with mapped name `mname` translates into a service by the name `mname_intf`.
- The service interface that is derived from the business interface of an EJB 3.x session bean consists of all methods of the EJB business interface.
- The service interface is remotable if it is derived from a remote business interface. Each EJB 3.x remote reference of each session bean within the enterprise application is exposed as an SCA reference. If the remote reference has the name `ref` and the name of the session bean is `beanname`, the SCA reference name is `beanname_ref`. Services thus derived can be invoked or wired to like an SCA service.

Procedure

1. Create a component in an SCA composite definition. Specify the `implementation.jee` component type and set the `archive` attribute to the name of the asset object.

   For example, the composite definition for a `implementation.jee` component named `ImplJeeComponent` that uses the `MyJEE.ear` archive resembles the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"; autowire="false"
    name="ImplJeeSCAComposite"
    targetNamespace="' class='inlinelink' target="_blank">http://jee">
  <component name="ImplJeeComponent">
    <implementation.jee archive="MyJEE.ear"/>
  </component>
</composite>
```

2. Optional: Write client code that looks up and invokes a derived SCA service.

   A client can look up and invoke a derived SCA service. For example, suppose the `ImplJeeComponent` component has a stateless session bean named `Bean1` with a remote interface `Intf1` in an EJB module in the enterprise application. Invoke the service with code such as the following:

   ```
   CompositeContext compositeContext = CurrentCompositeContext.getContext();
   Intf1 myEjb = compositeContext.getService(Intf1.class, "ImplJeeComponent/Bean1_Intf1");
   retStr = ejb.myOperation();
   ```

3. Optional: Define a reference to a derived service in an SCA component composite. Use the injected value of a matching @Reference SCA annotation to invoke an EJB service as an SCA service.

   Instead of writing client code to look up and invoke the derived SCA server (step 2), you can define a reference to the derived service in an SCA component composite. An SCA service in a component can use the injected value of a matching @Reference SCA annotation to invoke the EJB service as a SCA service.

   For example, the `ImplJeeComponent` component might invoke an EJB service as a SCA service using the following code. The SCA component reference code resembles:

   ```
   <reference name="MyRef" target="ImplJeeComponent/Bean1_Intf1>
   ```

   The SCA service code resembles:

   ```
   @Reference public Intf1 MyRef;
   .....
   String retStr = MyRef.myOperation();
   ....
   ```

Results

The enterprise application is available as an SCA component type. Any SCA component can refer to the derived services.

What to do next

Deploy the enterprise application and the SCA composite that uses the application as composition units of the same business-level application. You can use the administrative console or wsadmin commands to create the business-level application and add the enterprise application and the SCA composite as composition units.

The product provides the HelloJee sample. The sample has a non-SCA enhanced EJB. The Example section in the Using existing Java EE modules and components as SCA implementations topic summarizes how to deploy the HelloJee sample and access the non-SCA enhanced EJB.

Using SCA enhanced Java EE applications as SCA component implementations

You can use an existing Java Platform, Enterprise Edition (Java EE) application as a Service Component Architecture (SCA) component after enhancing the application. Define an SCA composite that has components with `implementation.ejb` and `implementation.web` component types which refer to Enterprise JavaBeans (EJB) and web modules within the Java EE application, and then promote services and references.

Before you begin

Identify the Java EE application that contains business logic in EJB or web modules to enable in the SCA environment.

A Java EE application is also called an enterprise application or enterprise archive (EAR) file.

About this task

The SCA programming model supports Java EE integration. You can expose EJB stateless session beans as SCA services by enabling an existing enterprise application module to be recognized as an SCA component and participate in an SCA composite. You can rewire the SCA services over different bindings. You can rewire EJB references in EJB and web modules to SCA references. Also, you can use SCA annotations to enable Java EE components such as stateless session beans, message driven beans, servlets, listeners, filters, and JavaServer Pages (JSP) files to consume SCA services and properties.

Define a composite in a file named `application.composite` and place the file in the enterprise application `META-INF` directory.

Procedure

1. Create a file named `application.composite` that defines components with component types of web and EJB modules of the enterprise application. Save the file to the enterprise application `META-INF` directory.

   Use the `implementation.web` element to declare a service component that is implemented by the web component. The component contains information for the annotations. To configure a web component implementation, use the following schema:

   ```
   <implementation.web web-uri="web_module_name"/>
   ```

   Use the `implementation.ejb` element to declare a service component that is implemented by a session bean component. To configure an EJB component implementation, use the following schema:

   ```
   <implementation.ejb ejb-link="EJB_module_name"/>
   ```

   The components can contain one or more supported elements, such as `<service>`, `<reference>`, and `<property>` elements. In the composite definition, promote the services and references to be available in the SCA component using the enterprise application as the component type.

   An example `application.composite` file for an enterprise application named `MyEAR.ear` follows. The file contains an EJB module named `myEJB.jar` with a bean named `MyBean` and a remote business interface named `MyRemoteIntf`. The file also contains a Web module named `myWeb.war`.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"; autowire="false" name="MyComposite"
           targetNamespace="' class='inlinelink' target="_blank">http://jee">

     <service name="MyBean_MyRemoteIntf" promote="MyEJBComponent/MyBean_MyRemoteIntf"/>
     <reference name="MySCAReference1" promote="MyEJBComponent/MySCAReference1"/>
     <reference name="MySCAReference2" promote="MyWebComponent/MySCAReference2"/>

     <component name="MyEJBComponent">
       <implementation.ejb ejb-link="MyEJB.jar#MyBean"/>
       <service name="MyBean_MyRemoteIntf">
         <interface.java ... />
       </service>
       <reference name="MySCAReference1" target="MySCAComponent1">
     </component>

     <component name="MyWebComponent">
       <implementation.web web-uri="MyWeb.war"/>
       <reference name="MySCAReference2" target="MySCAComponent2">
     </component>
   </composite>
   ```

2. Create a component in an SCA composite definition that has the `implementation.jee` component type and the `archive` attribute set to the name of the enterprise application asset object.

   For the example enterprise application `MyEAR.ear`, create a component such as the following:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"; autowire="false" name="ImplJeeSCAComposite"
         targetNamespace="' class='inlinelink' target="_blank">http://jee">
     <component name="ImplJeeComponent">
       <implementation.jee archive="MyEAR.ear"/>
     </component>
   </composite>
   ```

3. Inject the promoted references into @Reference annotations in the EJB and web modules. Also, rewire the promoted service, and reference the service from other SCA components.

   For information about using a composite file to support annotations, refer to the following topics:
   - Using SCA annotations with web modules
   - Using SCA annotations with session beans
   - Using SCA annotations with message-driven beans
4. Using a client or an SCA service, invoke the promoted EJB as an SCA service.

Results

The enterprise application is available as an SCA component type. Any SCA component can refer to the promoted services. Java EE modules can access SCA services and properties using SCA annotations. You can rewire the services and reference them like any other SCA service.

What to do next

Deploy the enterprise application and the SCA composite that uses the application as composition units of the same business-level application. You can use the administrative console or wsadmin commands to create the business-level application and add the enterprise application and the SCA composite as composition units.

The product provides the HelloJee sample. The sample has an SCA enhanced EJB. The Example section in the Using existing Java EE modules and components as SCA implementations topic summarizes how to deploy the HelloJee sample and access the SCA enhanced EJB.

Using SCA annotations with web modules

Use Java annotations for Service Component Architecture (SCA) to identify existing Java Platform, Enterprise Edition (Java EE) components, such as web modules, as SCA components that are a part of an SCA composite.

Before you begin

Identify and obtain the web module that represents your business logic that you want to enable within an SCA environment.

About this task

The SCA programming model supports Java EE integration. As a result, you can take advantage of SCA annotations to enable Java EE web components such as servlets, filters, and event listeners to consume SCA services. By using Java annotations that apply to SCA, you can enable existing web modules to be recognized as an SCA component and participate in an SCA composite.

Web modules can participate in SCA assembly as the implementation type of a component that does not offer services, even though you can configure or wire the component to other services. You can configure a web module with annotations to acquire references to services that are wired to the component by the SCA assembly. You can also use annotations when you want to obtain the value of a property using the @Property annotation, to inject a handle to the SCA component context using the @Context annotation or to obtain the component name using the @ComponentName annotation.

For a list of supported annotations for web modules, see the SCA specifications and APIs documentation.

You can also obtain SCA references in JavaServer Pages (JSP) files by using SCA JSP tag libraries. The following example illustrates how annotations are used within a JSP tag handler:

```
<%@ taglib uri="http://www.osoa.org/sca/sca_jsp.tld" prefix="sca" %>
<sca:reference name="service" type="test.MyService"  />
<% service.sayHello(); %>
```

Procedure

1. Add SCA annotations to the components that you want within a web module. Based on your needs, use the supported annotations to inject SCA information into your web module.
2. Define a component in the `application.composite` file in the `META-INF` directory.

   The `implementation.web` element is used to declare a service component that is implemented by the web component. The component contains information for the annotations. To configure this component implementation, use the following schema:

   ```
   <implementation.web web-uri="<module name>"/>
   ```

   For example, you can define a component in the EAR `META-INF/application.composite` file as follows:

```
<component name="WebAnnotationTestServletComponent">
   <implementation.web web-uri="SCA_JEE_InjectionWeb.war"/>
   <reference name="getServerDateReference" target="GetServerDateServiceComponent">
       <interface.java interface="sca.injection.test.GetServerDateService"/>
   </reference>
</component>
```

   The following example illustrates a servlet in the web application archive (WAR) module using SCA annotations:

```
 public class WebAnnotationTestServlet extends HttpServlet {
     @Reference GetServerDateService getServerDateReference;
 }
```

   The following example illustrates a JSP using SCA annotations by importing the SCA tag library:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://www.osoa.org/sca/sca_jsp.tld" prefix="sca" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN
" "http://www.w3.org/TR/html4/loose.dtd">

<sca:reference name="getServerDateReference"
type="sca.injection.test.GetServerDateService" />
<html>
   <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>TESTING SCA ANNOTATION IN JSP</title>
   </head>
   <body>
   <h2> Reference Annotation in JSP: </h2>
     GetServerDateService.getString returns : <%=getServerDateReference.getString() %>
   </body>
</html>
```

Results

You now have SCA-enabled Java EE web modules that take advantage of the SCA programming model.

What to do next

Deploy the components to a business-level application.

Using SCA annotations with session beans

Use Java annotations for Service Component Architecture (SCA) to identify existing Java Platform, Enterprise Edition (Java EE) components, such as session beans, as SCA components that are a part of an SCA composite.

Before you begin

Identify and obtain the session beans that represent your business logic that you want to enable within an SCA environment.

About this task

The SCA programming model supports Java EE integration. As a result, you can take advantage of SCA annotations to enable Java EE web components such as session beans to consume SCA services. By using Java annotations that apply to SCA, you can enable existing session beans to be recognized as an SCA component and participate in an SCA composite. Within Java EE environments, sessions beans encapsulate business logic to manage security, transactions, and remotable interfaces. Because service components within SCA environments play a similar role, you can take advantage of the capability to use session beans as a service component implementation in a Java EE environment.

Any session bean that serves as the implementation type of an SCA service component can use annotations to obtain an interface to the SCA services that are wired to the component by the SCA assembly. You can also use annotations when you want to obtain the value of a property using the @Property annotation, to inject a handle to the SCA component context using the @Context annotation or to obtain the component name using the @ComponentName annotation. When using SCA annotations, you must apply the injection using annotations after the session bean instance is created, but before invoking business methods on the bean instance.

For a list of supported annotations for session beans, see the SCA specifications and APIs documentation.

Procedure

1. Add SCA annotations to the components that you want within your session beans to enable the component to be recognized as an SCA component. Based on your needs, use the supported annotations to inject SCA information such as context or component name into your session beans.
2. Edit the `application.composite` in the `META-INF` directory of the Java EE JAR file.

   Define a component within the `application.composite` file whose implementation is defined by an `implementation.ejb` element and specifies a session bean within the module. You can define multiple components, each with an `implementation.ejb` link that points to a session bean.

   Because message-driven beans and session beans are enterprise beans, you can uniquely refer to both bean types in an ejb-link element.

   The `implementation.ejb` element is used to declare a service component that is implemented by the session bean component. The component contains information for the annotations. To configure this component implementation, use the following schema:

   ```
   <implementation.ejb ejb-link="<ejb_link_name>"/>
   ```

   The enterprise bean that serves as the component implementation is uniquely identified by the *<ejb_link_name>* attribute. The format of the *<ejb_link_name>* attribute is identical to the format of the ejb-link element in a Java EE deployment descriptor.

   If the Java EE archive that contains the composite file is an application enterprise archive (EAR) file, multiple session beans might have the same name. In this case, the value of the ejb-link element must be composed of a path name. The path name specifies the ejb-jar that contains the referenced enterprise bean with the value of the ejb-name of the referenced enterprise bean appended and separated from the path name with the # symbol. The path name is relative to the root of the EAR file. For the case where the Java EE archive is a JAR file for the Enterprise JavaBeans (EJB) module, omit the path name.

   For example, you can have a JAR module that has the following component defined in the `application.composite` file:

```
<component name="AnnotationTest">
    <implementation.ejb ejb-link="SCA_JEE_Injection.jar#AnnotationTest"/>
        <property name="property" type="xsd:string">Right</property>

    <reference name="getServerDateReference" target="GetServerDateServiceComponent">
    <interface.java interface="sca.injection.test.GetServerDateService"/>
    </reference>

</component>
```

In this example, the session bean, AnnotationTest, is consuming an SCA service exposed by the GetServerDataServiceComponent reference.

The AnnotationTest session bean includes the @Property, @Reference, @ComponentName, and @Context annotations:

```
import javax.ejb.Stateless;

import org.osoa.sca.ComponentContext;
import org.osoa.sca.annotations.ComponentName;
import org.osoa.sca.annotations.Context;
import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;

import sca.injection.test.GetServerDateService;
/**
* Session Bean implementation class AnnotationTest
*/
@Stateless
public class AnnotationTest implements AnnotationTestRemote, AnnotationTestLocal {


@Context protected ComponentContext myContext;
@ComponentName String myCompName;
@Property protected String property;
@Reference GetServerDateService getServerDateReference;

@Reference
GetServerDateService referenceDefault;

/**
* Default constructor.
*/
public AnnotationTest() {
// TODO Auto-generated constructor stub
}

public String getServString(){
GetServerDateService svc = myContext.getService(GetServerDateService.class, "getServerDateReference");
return svc.getString();
}

public String getProperty(){
return property;
}
public String getComponentName(){
return myCompName;
}
public String getReferenceDefault() {
if ( referenceDefault == null )
return null;
else
return referenceDefault.getString();
}

}
```

Results

You now have SCA-enabled Java EE session beans that take advantage of the SCA programming model.

What to do next

Deploy the components to a business-level application.

Using SCA annotations with message-driven beans

Use Java annotations for Service Component Architecture (SCA) to identify existing Java Platform, Enterprise Edition (Java EE) components, such as message-driven beans, as SCA components that are a part of an SCA composite.

Before you begin

Identify and obtain the message-driven beans that represent your business logic that you want to enable within an SCA environment.

About this task

The SCA programming model supports Java EE integration. As a result, you can take advantage of SCA annotations to enable Java EE web components such as message-driven beans to consume SCA services. By using Java annotations that apply to SCA, you can enable existing message-driven beans to be recognized as an SCA component and participate in an SCA composite.

Message-driven beans can only participate in SCA assembly as the implementation type of a component that does not offer services, even though you can configure or wire the component to other services. Because of the association with endpoints that are not controlled by SCA such as Java Message Service (JMS), do not instantiate message-driven beans arbitrarily. You must not use a message-driven bean as a service component implementation more than one time within the SCA assembly of the application package.

You can configure a message-driven bean that is defined as an implementation type of an SCA component with annotations in order to obtain references to services that are wired to the component by the SCA assembly by using the @Reference annotation. You can also use annotations when you want to obtain the value of a property using the @Property annotation, to inject a handle to the SCA component context using the @Context annotation or to obtain the component name using the @ComponentName annotation.

For a list of supported annotations for message-driven beans, see the SCA specifications and APIs documentation.

Procedure

1. Add SCA annotations to the components that you want within your message-driven beans. Based on your needs, use the supported annotations to inject SCA information into your message-driven beans.
2. Edit the `application.composite` in the META-INF directory of the Java EE JAR file.

   Define a component within the `application.composite` whose implementation is defined by an `implementation.ejb` element and specifies a message-driven bean within the module. You can define multiple components, each with an `implementation.ejb` link that points to a message-driven bean.

   Because message-driven beans and session beans are enterprise beans, you can uniquely refer to both bean types in an ejb-link element.

   The `implementation.ejb` element is used to declare a service component that is implemented by the message-driven bean component. The component contains information for the annotations. To configure this component implementation, use the following schema:

   ```
   <implementation.ejb ejb-link="<ejb_link_name>"/>
   ```

   The enterprise bean that serves as the component implementation is uniquely identified by the *<ejb_link_name>* attribute. The format of the *<ejb_link_name>* attribute is identical to the format of the ejb-link element in a Java EE deployment descriptor.

   If the Java EE archive that contains the composite file is an application enterprise archive (EAR) file, it is possible that multiple message-driven beans have the same name. In this case, the value of the ejb-link element must be composed of a path name specifying the ejb-jar that contains the referenced enterprise bean with the ejb-name of the referenced enterprise bean appended and separated from the

path name with the # symbol. The path name is relative to the root of the EAR file. For the case where the Java EE archive is a JAR file for the Enterprise JavaBeans (EJB) module, omit the path name.

For example, you can have a JAR module that has the following component defined in the `application.composite`:

```
<component name="AnnotationTest">
     <implementation.ejb ejb-link="SCA_JEE_Injection.jar#AnnotationTest"/>
                <property name="property" type="xsd:string">Right</property>

      <reference name="getServerDateReference" target="GetServerDateServiceComponent">
     <interface.java interface="sca.injection.test.GetServerDateService"/>
      </reference>

</component>
```

In the following example, the message-driven bean, AnnotationTestMDB, is consuming an SCA service exposed by the GetServerDataServiceComponent reference. The AnnotationTestMDB message driven bean includes the @Property and @Reference annotations.

```
@MessageDriven
public class AnnotationTestMDB implements MessageListener {

//Property Annotations
@Property protected String property;


//Reference Annotation
@Reference protected GetServerDateService getServerDateReference;



/**
* Default constructor.
*/
public AnnotationTestMDB() {
// TODO Auto-generated constructor stub
}

private static final String JMSCF_JNDI_NAME = "jms/AnnotationQueueFactory";
private static final String JMSResponseQ_JNDI_NAME = "jms/AnnotationResponseQueue";
/**
* @see MessageListener#onMessage(Message)
*/
public void onMessage(Message message) {
String strDefaultReference = null;

System.out.println("Inside onMessage()");
try {
System.out.println("onMessage: " + "Exercising annotations");
if ( getServerDateReference != null)
strDefaultReference = getServerDateReference.getString();
}
catch (RuntimeException e) {
strError = "Error - Failed WebAnnotationTestServlet.service()!";
e.printStackTrace();
}

if (strError != null){
System.out.println("onMessage: " + "Encountered an error while annotation work");
outSB.append("@FINALERROR" + strError);
} else {
System.out.println("onMessage: " + "Annotations successful: now creating reply message");
outSB.append("@PropertyDefault:" + propertyDefault);
outSB.append("@ReferenceDefault:" + strDefaultReference);
}
}
}
```

Results

You now have SCA-enabled Java EE message-driven beans to take advantage of the SCA programming model.

What to do next

Deploy the components to a business-level application.

SCA annotations

By using Java annotations that apply to Service Component Architecture (SCA), you can enable an existing Java Platform, Enterprise Edition (Java EE) component to be recognized as an SCA component and participate in an SCA composite.

The SCA programming model supports Java EE integration. As a result, you can take advantage of SCA annotations to enable the Java EE components, such as session beans, message driven beans, or web components to consume SCA services.

The target for annotations applies for these Java objects:

- Types such as a Java class, enum or interface
- Methods
- Fields representing local instance variables within a Java class
- Parameters within a Java method

Annotations supported by SCA are listed in the following table:

Table 91. SCA annotations. The annotations enable Java EE components.

Annotation class	Annotation	Properties
`org.osoa.sca.annotations.ComponentName.class`	The **@ComponentName** annotation specifies the injection of a component name.	• Annotation target: Method or field • There are no properties on the @ComponentName annotation.
`org.osoa.sca.annotations.Context.class`	The **@Context** annotation specifies to inject SCA context into a service component instance. When you inject a composite context for the component, the type of context is defined by the type of the Java class field or type of the setter method input argument. Specify the type as either ComponentContext or RequestContext.	• Annotation target: Method or field • There are no properties on the @Context annotation.
`org.osoa.sca.annotations.Property.class`	The **@Property** annotation specifies to inject configuration properties from service component configuration. You can inject a simple Java type or a complex Java type. The type of the property injected is defined by the type of the Java class field or type of the setter method input argument. You can use this annotation on protected or public fields, on setter methods, or on a constructor method.	• Annotation target: Method or field • Properties: **- name** The name of the property. The default is the name of the field of the Java class. This property is optional. (String) **- required** Specifies whether injection is required. The default value is `false`. This property is optional. (Boolean)
`org.osoa.sca.annotations.Reference.class`	The **@Reference** annotation specifies the injection of a service reference. The interface of the service injected is defined by the type of the Java class field or the type of the setter method input parameter.	• Annotation target: Method or field • Properties: **- name** The name of the property. The default is the name of the field of the Java class. This property is optional. (String) **- required** Specifies whether injection is required. The default value is `false`. This property is optional. (Boolean)

Rewiring EJB references to SCA references

You can rewire an Enterprise JavaBeans (EJB) reference to a Service Component Architecture (SCA) reference that provides the same named operations. The rewiring does not require a change to existing code.

Before you begin

Identify the Java Platform, Enterprise Edition (Java EE) application that contains business logic in EJB or web modules to enable in the SCA environment. A Java EE application is also called an enterprise application or enterprise archive (EAR) file.

Identify the SCA components and services to replace the EJB references in EJB and web modules.

About this task

The SCA programming model supports Java EE integration. You can rewire EJB references to SCA references without changing existing code that uses the EJB reference. Thus, you can use the SCA programming model without changing the implementation code.

Procedure

1. Create a file named `application.composite` that defines components with component types of web and EJB modules of the enterprise application. Save the file to the enterprise application `META-INF` directory.

   Use the `implementation.web` element to declare a service component that is implemented by the web component. The component contains information for the annotations. To configure a web component implementation, use the following schema:

```
<implementation.web web-uri="web_module_name"/>
```

   Use the `implementation.ejb` element to declare a service component that is implemented by a session bean component. To configure an EJB component implementation, use the following schema:

```
<implementation.ejb ejb-link="EJB_module_name"/>
```

   To rewire an EJB reference to an SCA service in an SCA component, add a `<reference>` element to the `jee` module component. For example, to rewire an EJB reference named `MyEJBRef` to an SCA service named `MySCASvc` in an SCA component named `MyComponent`, the `<reference>` element resembles the following:

```
<reference name="MyEJBRef" target="MyComponent/MySCASvc">
  <interface.java .... />
</reference>
```

   The reference names must match the name of the EJB reference. The contract of operations supported must also match. Because references must be promoted in the composite, the names of the references must be unique. After the EJB reference is promoted, all matching instances of the EJB reference in the module are rewired to the SCA service.

2. Create a component in an SCA composite definition that has the `implementation.jee` component type and the `archive` attribute set to the name of the enterprise application asset object.

   For an enterprise application named `MyEAR.ear`, create a component such as the following:

```
<component name="ImplJeeComponent">
  <implementation.jee archive="MyEAR.ear"/>
</component>
```

Results

The EJB references are rewired to SCA references enabling Java EE modules to use SCA features without changing its implementation.

What to do next

Deploy the enterprise application and the SCA composite that uses the application as composition units of the same business-level application. You can use the administrative console or wsadmin commands to create the business-level application and add the enterprise application and the SCA composite as composition units.

Using OSGi applications as SCA component implementations

You can use an OSGi application as a Service Component Architecture (SCA) component.

Before you begin

Identify the OSGi application to use as an SCA component. An OSGi application is a collection of OSGi bundles that use the Blueprint component model to expose or consume services. An OSGi application contains an application manifest that declares the following services:

- Services that the application provides that can be accessed from outside the application
- Services that the application wants to consume from outside the application

You can use SCA to provide service bindings for these services.

About this task

To provide service bindings using SCA, do the following:

1. Modify an OSGi application to provide or use remote services.
2. Write an SCA composite definition that uses an OSGi application as a component implementation, and provide bindings for its remote services.

An OSGi application declares external services in the Application-ImportService and Application-ExportService statements of its application manifest, which is provided in the `META-INF/APPLICATION.MF` file of the enterprise bundle archive (EBA). The Application-ExportService statement declares remote services that are provided by the OSGi application. The Application-ImportService statement declares services on which the OSGi application depends. All services specified in the application manifest are remotable.

Procedure

1. Create an `implementation.osgiapp` component in an SCA composite definition.

   Specify the `implementation.osgiapp` component type and set `applicationSymbolicName` and `applicationVersion` to values that match the `Application-SymbolicName` and `Application-Version` attributes in the application manifest.

   For example, suppose the OSGi application manifest, the `META-INF/APPLICATION.MF` file, contains the following headers:

   ```
   Application-SymbolicName: helloworldApp
   Application-Version: 1.0.0
   ```

   Configure a component that references these headers in an SCA composite definition; for example:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <composite
     xmlns="http://www.osoa.org/xmlns/sca/1.0"
     xmlns:scafp="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
     targetNamespace="http://www.example.com"
     name="HelloWorldComposite">
       <component name="HelloWorldComponent">
           <scafp:implementation.osgiapp
              applicationSymbolicName="helloworldApp"
              applicationVersion="1.0.0"/>
       </component></composite>
   ```

The `implementation.osgiapp` element requires the use of an XML namespace prefix that is associated with the `"http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"` namespace.

The `applicationVersion` attribute is optional. You can add it to ensure that a particular version of an OSGi application is used.

2. Identify Blueprint services to be made available remotely.
   a. Edit the Application-ExportService header in the OSGi application manifest so that it identifies one or more service interfaces to be exported.

      The following example header specifies that Blueprint services which implement and export the `example.HelloWorld` interface are to be made available outside the application.

      ```
      Application-ExportService: example.HelloWorld
      ```

      An example of a Blueprint component with such a service follows:

      ```
      <blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
      <bean id="helloWorldComponent" class="example.HelloWorldImpl"
        <service id="helloWorld" ref="helloWorldComponent"
           interface="example.HelloWorld">
         <service-properties>
           <entry key="service.exported.interfaces" value="*"/>
         </service-properties>
        </service>
      </blueprint>
      ```

      The Blueprint service must specify the `service.exported.interfaces` property to identify which of its interfaces are to be exposed remotely. The value can be an asterisk (`*`) to indicate that all of its interfaces are available remotely, or it can be a particular interface name.

   b. Configure an SCA service that corresponds to each remotable Blueprint service in the component.

      Use the Blueprint service `id` value for the SCA service name. If a Blueprint service does not have an `id` value, use the bean `id` value instead. If more than one interface is defined for the Blueprint service, for the first service interface SCA Service name use the Blueprint `id` value. For the second and later services, use the *`Blueprint id_ fully qualified interface name`* value for the SCA service name in the order of interfaces defined in the `blueprint.xml` file.

      For example, add the `helloWorld` service that is shown in the Blueprint component from step 2a to the SCA composite definition from step 1:

      ```
      <?xml version="1.0" encoding="UTF-8"?>
       <composite
        xmlns="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:scafp="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
        targetNamespace="http://www.example.com"
        name="HelloWorldComposite">
          <component name="HelloWorldComponent">
              <scafp:implementation.osgiapp
                 applicationSymbolicName="helloworldApp"
                 applicationVersion="1.0.0"/>
              <service name="helloWorld">
                  <binding.sca>
              </service>
          </component>
      </composite>
      ```

      The example uses `binding.sca` for the service binding. The service can be made available over one or more of the other bindings that SCA supports, except the EJB 2.x service binding.

      The SCA service element is not required. If it is not specified, the service is made available over `binding.sca` by default.

3. Identify services to be provided from outside the OSGi application.
   a. Edit the Application-ImportService header in the OSGi application manifest so that it identifies one or more service interfaces to be imported.

      The following example header specifies that the `example.Greeting` service interface be imported:

      ```
      Application-ImportService: example.Greeting
      ```

A Blueprint reference explicitly requests an imported service by filtering for the `service.imported` property. Remote services use pass-by-value semantics instead of pass-by-reference semantics for local services within the application. The following example shows a Blueprint component with such a reference:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="helloWorldComponent" class="example.HelloWorldImpl"
    <property name="greetingList" ref="greetingRef"/>
  </bean>
  <service id="helloWorld" ref="helloWorldComponent"
     interface="example.HelloWorld">
   <service-properties>
     <entry key="service.exported.interfaces" value="*"/>
   </service-properties>
  </service>
  <reference-list id="greetingRef" interface="example.Greeting"
     filter="(&amp;(service.imported=*))"/>
</blueprint>
```

b. Configure an SCA reference that corresponds to each imported service in the component.

Use the fully qualified interface name for the SCA reference name.

For example, add the `example.Greeting` reference that is shown in the Blueprint component from step 3 to the SCA composite definition from step 2:

```
<?xml version="1.0" encoding="UTF-8"?>
 <composite
  xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:scafp="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
  targetNamespace="http://www.example.com"
  name="HelloWorldComposite">
    <component name="HelloWorldComponent">
        <scafp:implementation.osgiapp
           applicationSymbolicName="helloworldApp"
           applicationVersion="1.0.0"/>
        <service name="helloWorld">
            <binding.sca>
        </service>
        <reference name="example.Greeting">
            <binding.sca uri="MyGreetingComponent">
        </reference>
    </component>
</composite>
```

The example uses `binding.sca` for the reference binding. The reference can use one or more of the other bindings that SCA supports.

An SCA reference for an `implementation.osgiapp` component implicitly has 0-to-many for the multiplicity attribute (`multiplicity='0..n'`). This means you can wire the reference to 0, 1, or multiple services depending upon the requirements of the application. As to the multiplicity attribute:

- A blueprint `reference-list` element selects multiple services. The blueprint implementation cannot make any assumptions about the order of the services in the reference list compared to the order of bindings in the composite definition.
- A blueprint `reference` element selects a single service. If the SCA reference provides more than one binding, the selection of which binding is used is unspecified. If the SCA reference does not provide any bindings, the dependency is unsatisfied. If the reference is mandatory, the bean might not start.

You can override the multiplicity attribute on the SCA reference, setting 1-to-many (`'1..n'`) or 1-to-1 (`'1..1'`), to ensure that a particular number of bindings is specified.

Results

The OSGi application is defined as an SCA component.

What to do next

Deploy the OSGi application and the SCA composite that uses the application as composition units of the same business-level application. You can use the administrative console or wsadmin commands to create

a business-level application, import the EBA file and SCA composite as assets, and then add the EBA and SCA assets as composition units to the business-level application.

SCA programming model support in OSGi applications

This topic describes the features of the Service Component Architecture (SCA) programming model that can be used with OSGi applications.

- Annotations
- Authorization policy
- Mapping of Blueprint transaction metadata to SCA service and reference transaction intents
- Using intents in an OSGi application
- Interface definition
- Custom wire format and operation selector
- Service Data Objects (SDO) version 2.1.1 with OSGi applications

Annotations

The following annotations from the org.osoa.sca.annotations package are supported within the interface classes in an OSGi application:

- @OneWay
- @Remotable

  When an OSGi application is used as an SCA component, its service and reference interfaces are automatically treated as remotable. The @Remotable annotation is not required. However, if you want to use the same Java source interface in other contexts, it might need to contain the @Remotable annotation:

  – If you copy the interface to the SCA asset and specify it in an `interface.java` element for an `implementation.osgiapp` service or reference, the interface must contain the @Remotable annotation. Otherwise, an interface incompatibility error occurs.

  – If you wire `implementation.java` and `implementation.osgiapp` components to each other, the interface must contain the @Remotable annotation. Otherwise, the `implementation.java` component considers the interface to be local and an interface incompatibility error occurs.

No annotations from the org.osoa.sca.annotations package are supported within the implementation classes in an OSGi application.

Authorization policy

You can attach an SCA policy set containing authorization policy statements to an `implementation.osgiapp` component. The policy set applies to all services of the component. It applies only when the services are started through SCA service bindings, and not when the OSGi application internally uses its services.

SCA does not support the use of the org.osoa.sca.annotations.PolicySet annotation or the annotations in the javax.annotation.security package in Blueprint implementation classes.

The configuration of role-based security for SCA components is independent of the configuration of role-based security for a Web application bundle (WAB). In other words, the roles and role mappings used for SCA components and for WABs are separate.

Mapping of Blueprint transaction metadata to SCA service and reference transaction intents

Use the transaction metadata in the Blueprint component definition to define the transactional environment of an `implementation.osgiapp` component

You can use the propagatesTransaction, suspendsTransaction, transactedOneWay, and exactlyOnce intents with the services and references of an `implementation.osgiapp` component. Use the correct intents for the transaction metadata in the Blueprint component definition. SCA does not detect mismatches. The following table shows valid combinations of Blueprint transaction metadata and transaction intents.

Table 92. Valid combinations of transaction intents in OSGi applications. Use valid transaction intents for services and references of an `implementation.osgiapp` component.

Transaction definition in Blueprint component	**Valid service intents**	**Valid reference intents**
Required	• propagatesTransaction • transactedOneWay • exactlyOnce	• propagatesTransaction • suspendsTransaction • transactedOneWay
Mandatory	• propagatesTransaction • transactedOneWay • exactlyOnce	• propagatesTransaction • suspendsTransaction • transactedOneWay
RequiresNew	suspendsTransaction	• propagatesTransaction • suspendsTransaction • transactedOneWay
Supports	propagatesTransaction	• propagatesTransaction • suspendsTransaction
NotSupported	suspendsTransaction	
Never		

The managedTransaction.global, managedTransaction.local, and noManagedTransaction intents do not apply for `implementation.osgiapp` components.

Not all service bindings support all intents. For information about each intent, consult the SCA Transaction Policy specification.

The SCA run time does not effect the propagatesTransaction or suspendsTransaction behavior on an SCA service. Use the correct transaction definition in the Blueprint component to achieve the wanted behavior. The main purpose of using an intent is to document a requirement to use a binding that supports it.

Using intents in an OSGi application

A Blueprint service can require a policy intent by using the `service.exported.intents` service property as shown in the following example:

```
<service ref="componentImplementation"
   interface="example.MyTransactionalServiceInterface">
   <service-properties>
      <entry key="service.exported.intents" value="propagatesTransaction"/>
   </service-properties>
</service>
```

Using the `service.exported.intents` property has the same effect as putting the intent in the SCA service definition.

The Application-ImportService header in the OSGi application manifest can filter for a particular policy intent on an imported service:

```
Application-ImportService: example.Greeting;filter="(&(service.intents=propagatesTransaction))"
```

If the SCA reference does not provide the intent, no services are imported.

Interface definition

It is not required to specify an interface element on a component service or component reference. You can specify an interface to enforce a given contract with the implementation. In this case, the interface must be compatible with the implementation as stated in the SCA specification. The SCA run time cannot validate that the interfaces are compatible until the business-level application is started.

If you specify an `interface.java` or `interface.wsdl` element, it must refer to a Java class or WSDL file that is packaged in the SCA asset, or that is imported from a shared asset.

Custom wire format and operation selector

The JMS binding enables you to provide classes that customize its function. The classes can provide:

- A wire format handler that transforms data between the application interface and a JMS Message
- A custom operation selector to determine the operation name from a JMS Message

You can package these classes in a bundle within the enterprise bundle archive (EBA). This is necessary to have access to Java packages inside the EBA bundles. For example, if a wire format handler needs to transform the content of a JMS Message into user-defined types defined within the EBA, the handler must be packaged in the EBA. To indicate that the handler is packaged in the EBA, you must use the `deferLoad` attribute; for example:

```
<composite
  xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:scafp="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
  targetNamespace="http://www.example.com"
  name="MyOSGiComposite">
    <component name="MyOSGiComponent">
      <scafp:implementation.osgiapp ...>
        <service name="MyOSGiService">
          <binding.jms>
            <activationSpec ...>
              <scafp:wireFormat.jmsCustom
                      class="example.WireHandler" deferLoad="true"/>
              <scafp:operationSelector.jmsCustom
                      class="example.OpSelector" deferLoad=="true"/>
          </binding.jms>
        </service>
    </component>
</composite>
```

Service Data Objects (SDO) version 2.1.1 with OSGi applications

You can use Service Data Objects version 2.1.1 over SCA default, Web service, JMS and HTTP bindings for an `implementation.osgiapp` component service. Package schema definition (XSD) files in an SCA asset, or import XSD files from a shared asset.

The following limitations apply when you use SDO with an `implementation.osgiapp` component:

- You must invoke the OSGi application over a supported SCA service binding, such as an SCA default, Web service, JMS or HTTP binding. In particular, you cannot use the SDO functionality in your OSGi application when you access the OSGi application over HTTP using the Web container access path that you typically use to access OSGi application code packaged in an OSGi Web application bundle (WAB).
- The product does not support the injection of the SCA-managed default HelperContext object using annotation. However, you can implement an application programming interface (API), which uses the commonj.sdo.helper.SDO class, to access the default HelperContext object. For more information about the HelperContext object, see the documentation about accessing default HelperContext objects in SCA applications
- You cannot use SDO in an OSGi application that is accessed over the SCA service bindings `binding.atom` or `binding.ejb`.

Note: If your application previously used SDO version 1.0.0, change the Import-Package to:

```
Import-Package: commonj.sdo;version="[2.0.0,3.0.0)"
```

Using Spring 2.5.5 containers in SCA applications

You can use the Service Component Architecture (SCA) programming model to invoke beans that follow the Java 2 Platform, Standard Edition (J2SE) programming model in a Spring 2.5.5 container. The product supports components implemented with Spring Framework that use `<implementation.spring>` in composite definitions.

Before you begin

Before you start the procedure in this topic, do the following:

1. Become familiar with the SCA Spring Component Implementation 1.0.0 specification and Spring 2.5.5 programming. You must use Spring 2.5.5 for the SCA composite component implementation. The product does not support other levels of Spring.

   The product only supports the Spring Framework using the J2SE programming model. Local Java Naming and Directory Interface (JNDI) lookups are not supported, thus the product does not support the Spring Framework using the Java Platform, Enterprise Edition (Java EE) 5 programming model. Any attempts by a Spring container to perform a local JNDI lookup fail.
2. Decide whether to use `implementation.spring` or `implementation.jee` for your Spring application.

   The procedure in this topic discusses using `implementation.spring` in an SCA component implementation. If a Spring application is, in fact, a specialized Java EE application, instead of using `implementation.spring`, you can use `implementation.jee` to define the SCA component implementation and enable the Spring application to be part of an SCA component.

   For example, if a Spring application is packaged in a web module (WAR), then you can use an enterprise archive (EAR) that contains the WAR file as a component implementation using `implementation.jee`. The Spring application works like before and the Java EE context is not affected. Spring beans are not available in an SCA domain; however, other modules that are compatible with `implementation.jee` within the EAR file can participate in SCA.

   For information on deploying a Spring application with `implementation.jee`, see the topic on using existing Java EE modules and components as SCA implementations.
3. If you decide to use `implementation.spring` and you want beans in your Spring application to perform JNDI lookups, set `com.ibm.websphere.naming.WsnInitialContextFactory` as a property for the beans.

   You can use globally defined JNDI resources, such as databases and connection factories, in a Spring bean definition with `implementation.spring`. However, you must set a Spring JNDI template bean that has been configured to reference the WebSphere initial context factory, `com.ibm.websphere.naming.WsnInitialContextFactory`, as a property for beans performing JNDI lookups. For example, the following bean definitions specify that the `myJndiResource` bean use the WebSphere initial context factory:

```
<bean id="myJndiResource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/myJndiResource"/>
  <property name="lookupOnStartup" value="true"/>
  <property name="cache" value="true"/>
  <property name="jndiTemplate" ref="wasJndiTemplate"/>
</bean>

<bean id="wasJndiTemplate" class="org.springframework.jndi.JndiTemplate">
  <property name="environment">
    <props>
      <prop key="java.naming.factory.initial">com.ibm.websphere.naming.WsnInitialContextFactory</prop>
    </props>
  </property>
</bean>
```

The SCA container for implementation.spring does not run in a Java EE container, so it does not have access to the Java EE local JNDI namespace. By specifying that your Spring beans use the WebSphere initial context factory, you can prevent java:comp/ext from being prepended to the JNDI name during lookup.

About this task

You can use a Spring application context as an implementation within an SCA composite component. Define the implementation using Spring 2.5.5. A Spring application context provides a complete composite, exposing services and using references using SCA.

A component that uses Spring for an implementation can wire SCA services and references without introducing SCA metadata into the Spring configuration. Generally, the Spring context knows little about the SCA environment.

Procedure

1. Define a component implementation that uses the Spring Framework in a composite definition.

   The Spring component implementation in a composite definition has the following format:

   ```
   <implementation.spring location="targetURI"/>
   ```

   The location attribute of the element specifies the target uniform resource indicator (URI) of a directory, or the fully qualified path that contains the Spring application context files.

   There are two ways that you can specify the target URI in the location attribute:

   - Specify a fully qualified path:

     ```
     <implementation.spring location="./spring/application-context.xml"/>
     ```

   - Specify a directory:

     ```
     <implementation.spring location="./spring"/>
     ```

     The target URI specifies the resource as a directory, here named spring, that has all the Spring-related files. To point to one application context, a META-INF/MANIFEST.MF file in that directory must contain a Spring-Context header of the format Spring-Context: *path*, where *path* is a relative path with respect to the location URI. For example:

     ```
     Spring-Context: META-INF/spring/application-context.xml
     ```

     If the META-INF/MANIFEST.MF file does not exist or does not contain the Spring-Context header, then the product builds an application context using the application-context.xml file in the META-INF/spring directory. If the META-INF/spring/application-context.xml file does not exist, then the application does not deploy.

2. Optional: Override implicit mapping of Spring resource to SCA resources.

   By default, SCA implicitly maps Spring resources to SCA resources as follows:

   - Each <bean/> becomes <sca:service/>.
   - Each unresolved <property/> becomes <sca:reference/> if typed by an interface.
   - Each unresolved <property/> becomes <sca:property/> if not typed by an interface.

   You can override this default mapping by using the SCA XML extensions in the Spring context to create explicit declarations. You typically do this to enable the Spring container to decorate the bean using, for example, Spring Aspect-Oriented Programming (AOP). The SCA XML extensions are defined as follows:

   - <sca:service/> defines each service exposed by the context.
   - <sca:reference/> defines each reference exposed by the context.
   - <sca:property/> defines each property exposed by the context.<sca:property/> must be a bean class and not a primitive type.

   For example, the properties checkingAccountService, calculatorService, and stockQuoteService defined in the following Spring configuration are declared explicitly as SCA beans in a Spring application context using the <sca:reference> element:

```
<beans>
  <bean id="AccountServiceBean" class="bigbank.account.AccountServiceImpl">
    <property name="calculatorService" ref="calculatorService"/>
    <property name="stockQuoteService" ref="stockQuoteService"/>
    <property name="checkingAccountService" ref="checkingAccountService"/>
    <!-- Some implicit references and properties follow. A property with a reference
         not satisifed * within the Spring application context. -->
    <property name="savingsAccountService" ref="savingsAccountService"/>
    <property name="stockAccountService" ref="stockAccountService"/>
    <property name="currency" value="EURO"/>
  </bean>
  <sca:reference name="checkingAccountService" type="bigbank.account.checking.CheckingAccountService"/>
  <ca:reference name="calculatorService" type="bigbank.calculator.CalculatorService"/>
  <sca:reference name="stockQuoteService" type="bigbank.stockquote.StockQuoteService"/>
</beans>
```

3. If you completed step 2, to use the SCA XML extensions you must add the SCA schema to the application context.

   Add the SCA schema, `http://www.osoa.org/xmlns/sca/1.0/spring-sca.xsd` , to the application context. Specify a Spring application context that defines the SCA schema namespace and makes the Spring application aware of the SCA-related beans; for example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sca="http://www.springframework.org/schema/sca"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/sca
         http://www.osoa.org/xmlns/sca/1.0/spring-sca.xsd">
  <sca:service name="StockQuoteService"
       type="bigbank.stockquote.StockQuoteService" target="StockQuoteServiceBean"/>
  <bean id="StockQuoteServiceBean" class="bigbank.stockquote.StockQuoteImpl">
  </bean>
</beans>
```

   For more information about defining an application context, see the reference documentation provided by the Spring source community. For more information about the SCA schema, see the SCA Spring Implementation specification.

4. Package the SCA application context file in your service Java archive (JAR) file at the location specified in your composite definition.

   For example, package the SCA application context file in a Spring service JAR file is named `helloworld-spring.jar`.

5. Create a Spring runtime JAR file that contains Spring and product SCA runtime files.

   Because the Spring runtime binary files are not shipped with the product, you must create an asset that contains three Spring framework JAR files and one product JAR file. Create the asset as follows:

   a. If you do not have the Spring 2.5.5 framework binary files, go to Spring Community Downloads at `http://www.springsource.com/download/community?project=` and download `spring-framework-2.5.5.zip`.

      After you download the compressed `.zip` file, extract it to a temporary directory.

   b. Create an empty directory; for example, `C:\SpringAsset`.

   c. Copy the Spring 2.5.5 framework binary files `spring-beans.jar`, `spring-context.jar`, and `spring-core.jar` to the empty directory.

      If the Spring application requires transaction support, copy all the Spring JAR files into the empty directory. The AspectJ Weaver library files are also required for transaction support.

   d. Copy the *app_server_root*`/optionalLibraries/sca/SCA-implementation-spring-runtime-1.0.1.jar` file to the same directory in which you placed the three Spring JAR files.

      Note: If you install a product maintenance package, update the `SpringSharedLibAsset.jar` with the newer version of the *app_server_root*`/optionalLibraries/sca/SCA-implementation-spring-runtime-1.0.1.jar` file.

e. Create a JAR file named `SpringSharedLibAsset.jar` that contains the four JAR files you placed in the `C:\SpringAsset` directory.

   For example, at a command prompt, enter the following commands:

   1) `cd C:\SpringAsset`
   2) `jar –cvf SpringSharedLibAsset.jar *.jar`

   For performances reasons, the JAR file must be named `SpringSharedLibAsset.jar`.

f. Verify that the `SpringSharedLibAsset.jar` file contains all four JAR files in the root directory.

6. Import the Spring runtime JAR file as an asset.

   To import `SpringSharedLibAsset.jar` as an asset using a wsadmin Jython command, enter the following command:

   ```
   AdminTask.importAsset('-source SpringSharedLibAsset.jar -storageType FULL')
   ```

   You can also use the administrative console.

7. Import the Spring service JAR file as an asset with a dependency on the Spring runtime asset.

   Suppose that your Spring service JAR file is named `helloworld-spring.jar`. Import your Spring service JAR file, `helloworld-spring.jar`, and create a dependency on the Spring runtime asset, `SpringSharedLibAsset.jar`. The dependency enables the product to access the necessary Spring classes. You can import the Spring service JAR file using the following wsadmin Jython command:

   ```
   AdminTask.importAsset('-source helloworld-spring.jar -storageType FULL -AssetOptions
    [[ helloworld-spring.jar helloworld-spring.jar "" "" "" spec=zip assetname=SpringSharedLibAsset.jar
    "" false]]')
   ```

   Alternatively, you can use the administrative console to import the asset with a dependency:

   a. On the administrative console, click **Applications** > **Application Types** > **Assets**.
   b. On the Assets page, click **Import**.
   c. On the Upload asset page, specify the full path name of the service JAR file and click **Next**.
   d. On the Select options for importing an asset page, specify a dependency.
      1) Click **Manage Relationships**. The button is near the bottom of the page. The product detects asset relationships automatically by matching the dependencies defined in the JAR manifest with the assets that are already imported into the administrative domain.
      2) Select your Spring runtime JAR file, such as `SpringSharedLibAsset.jar`, click **>>** to move the JAR file to the **Selected** list.
      3) Click **OK**.

         Under **Current asset relationships**, the asset name is shown; for example:

         ```
         WebSphere:assetname=SpringSharedLibAsset.jar
         ```

      4) Click **Next**.
   e. Click **Finish**.
   f. After the product imports the asset, click **Save**.

Results

The Spring runtime JAR file and Spring service JAR file are imported assets available for use in a business-level application. The Spring service JAR asset has a dependency on the Spring runtime asset.

What to do next

Add the JAR assets that you created to an SCA business-level application.

To learn about Spring implementation features that the product supports but which are not defined in the SCA Spring Implementation specification, see "Additional Spring component implementation features."

Additional Spring component implementation features

The product supports Spring implementation features that are not defined in the Service Component Architecture (SCA) Spring Implementation specification.

The product supports the following features:

- Constructor injection

  The product supports the injection of SCA references and properties within Spring bean constructors. Define `<constructor-arg>` elements that specify the appropriate type of the SCA references or properties to use. If the elements do not specify the `type` attribute, then at least specify the index attribute. The product only supports constructor injection when the Spring bean has a single constructor.

- Using the ClassPathXmlApplicationContext bean definition

  When the ClassPathXMLApplicationContext bean definition is in an application context, only the beans and properties defined in the top-level application context can be mapped, either explicitly or implicitly, to SCA resources.

  The product does not support using a FileSystemXMLApplicationContext definition to load an application context file.

- `<import>` elements in application context files

  The product supports use of `<import>` elements in application context files. Each `<import>` element points to an application context XML file; for example:

```
<bean>
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>
  <bean id="bean1" class="..."/>
  <bean id="bean2" class="..."/>
</beans>
```

Chapter 20. Developing Scheduler service

This page provides a starting point for finding information about the scheduler service, a WebSphere programming extension responsible for starting actions at specific times or intervals.

Schedulers are persistent and transactional timer services that run Enterprise JavaBeans methods or send Java Message Service messages using any Java Message Service messages using any Java Platform, Enterprise Edition (Java EE) server application.

The scheduler service helps minimize IT costs and increase application speed and responsiveness by maximizing utilization of existing computing resources.

The scheduler service provides the ability to reliably process workloads using parallel processing and schedule resource-intensive tasks to process during low traffic off-hours.

Developing and scheduling tasks

To develop and schedule tasks, use a configured scheduler.

Procedure

1. Look up a configured scheduler. Refer to the Accessing schedules topic. Each configured scheduler is available from two different programming models:
   - A Java Platform, Enterprise Edition (Java EE) server application, such as a servlet or Enterprise JavaBeans (EJB) component, can use the Scheduler API. Schedulers are accessed by looking them up using a Java Naming and Directory Interface (JNDI) name or resource reference.
   - Java Management Extensions (JMX) applications, such as wsadmin scripts, can use the Scheduler API using WASScheduler MBeans.
2. Develop the task.

   The Scheduler API supports different implementations of the TaskInfo interface, each of which can be used to schedule a particular type of work. Refer to the Developing a task that calls a session bean topic and Develop a task that sends a Java Message Service (JMS) message topic for details. The task object that is referenced in the Develop a task that sends a Java Message Service (JMS) message topic can send a JMS message to either a queue or a topic.

   Attention: Creating and manipulating scheduled tasks through the Scheduler interface is only supported from within the EJB container or Web container (Enterprise beans or servlets). Looking up and using a configured scheduler from a Java EE application client container is not supported.
3. Receive scheduler notifications. A notification sink is set on a task in order to receive the notification events that are generated by a scheduler when it performs an operation on the task.
4. Use custom calendars. You can assign a UserCalendar session bean to a task that allows schedulers to use custom and predefined date algorithms to determine when a task should run. Refer to the UserCalendar interface topic for details.
5. Submit tasks to a scheduler. After a TaskInfo object has been created, it can be submitted to the scheduler for task creation by calling the Scheduler.create() method.
6. Manage tasks with a scheduler.
7. Secure tasks with a scheduler.

Example

You can use the SIMPLE and CRON calendars from any Java EE application. This example illustrates the process.

- Using default scheduler calendars

Using default scheduler calendars. The following code examples illustrates how to use connections correctly and incorrectly.

Using default scheduler calendars involves looking up the default UserCalendarHome Enterprise JavaBeans (EJB) home object, creating the UserCalendar bean and calling the applyDelta() method. For details on the applyDelta method as well as the syntax for the SIMPLE and CRON calendars, see UserCalendar interface topic.

```
import java.util.Date;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import com.ibm.websphere.scheduler.UserCalendar;
import com.ibm.websphere.scheduler.UserCalendarHome;

// Create an initial context
InitialContext ctx = new InitialContext();

// Lookup and narrow the default UserCalendar home.
UserCalendarHome defaultCalHome=(UserCalendarHome)
 PortableRemoteObject.narrow(ctx.lookup(
      UserCalendarHome.DEFAULT_CALENDAR_JNDI_NAME),
    UserCalendarHome.class);

// Create the default UserCalendar instance.
UserCalendar defaultCal = defaultCalHome.create();

// Calculate a date using CRON based on the current
// date and time.  Return the next date that is
// Saturday at 2AM
Date newDate =
    defaultCal.applyDelta(new Date(),
        "CRON", "0 0 2 ? * SAT");
```

Accessing schedulers

Each configured scheduler is available using the Scheduler API from a Java Platform, Enterprise Edition (Java EE) server application, such as a servlet or Enterprise JavaBeans (EJB) module. Use a Java Naming and Directory Interface (JNDI). name or resource reference to access schedulers. Each scheduler is also available using the Java™ Management Extensions (JMX) API, using its associated WASScheduler MBean.

About this task

Scheduler and WASScheduler interfaces are the starting point for all scheduler activities. Each scheduler is independent and allows task life cycle operations, such as creating new tasks.

Procedure

1. Locate schedulers using the javax.naming.Context.lookup() method from a Java EE server application, such as a servlet or EJB module like the following example:

   ```
   //lookup the scheduler to be used
   import com.ibm.websphere.scheduler.Scheduler;
   import javax.naming.InitialContext;
   Scheduler scheduler = (Scheduler)new InitialContext.lookup("java:comp/env/sched/MyScheduler");
   ```

2. Use wsadmin to locate a WASScheduler MBean using JACL scripting:

   ```
   set jndiName sched/MyScheduler

   # Map the JNDI name to the mbean name.  The mbean name is
   # formed by replacing the / in the JNDI namewith . and prepending
   # Scheduler_
   regsub -all {/} $jndiName "." jndiName
   set mbeanName Scheduler_$jndiName
   ```

```
puts "Looking-up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
puts $sched
```

Results

The scheduler is now available to use from a Java EE server application or from a JMX API client. To create a task see the topics, Developing a task that calls a session bean or Developing a task that sends a JMS message.

Developing a task that calls a session bean

The Scheduler API and WASScheduler MBean API support different implementations of the TaskInfo interface, each of which can be used to schedule a particular type of work. This topic describes how to create a task to call a method on a TaskHandler session bean.

About this task

To create a task to call a method on a TaskHandler session bean, use these steps.

Procedure

1. Create a new enterprise application with an Enterprise JavaBeans (EJB) module. This application hosts the TaskHandler EJB module.
2. Create a stateless session bean in the EJB Module that implements the process() method in the com.ibm.websphere.scheduler.TaskHandler remote interface. Place the business logic you want created in the process() method. The process() method is called when the task runs. The Home and Remote interfaces must be set as follows in the deployment descriptor bean:
   - com.ibm.websphere.scheduler.TaskHandlerHome
   - com.ibm.websphere.scheduler.TaskHandler
3. Create an instance of the BeanTaskInfo interface by using the following example factory method. Using a JavaServer Pages (JSP) file, servlet or EJB component, create the instance as shown in the following code example. This code should coexist in the same application as the previously created TaskHandler EJB module:

   ```
   // Assume that a scheduler has already been looked-up in JNDI.
   BeanTaskInfo taskInfo = (BeanTaskInfo) scheduler.createTaskInfo(BeanTaskInfo.class)
   ```

   You can also use the wsadmin tool to create the instance as shown in the following JACL scripting example:

   ```
   set taskHandlerHomeJNDIName ejb/MyTaskHandler

   # Map the JNDI name to the mbean name.  The mbean name is formed by replacing the / in the jndi name
   # with . and prepending Scheduler_
   regsub -all {/} $jndiName "." jndiName
   set mbeanName Scheduler_$jndiName

   puts "Looking-up Scheduler MBean $mbeanName"
   set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
   puts $sched

   # Get the ObjectName format of the Scheduler MBean
   set schedO [$AdminControl makeObjectName $sched]

   # Create a BeanTaskInfo object using invoke_jmx
   puts "Creating BeanTaskInfo"
   set params [java::new {java.lang.Object[]} 1]
   $params set 0 [java::field com.ibm.websphere.scheduler.BeanTaskInfo class]

   set sigs [java::new {java.lang.String[]} 1]
   $sigs set 0 java.lang.Class
   ```

```
set ti [$AdminControl invoke_jmx $schedO createTaskInfo $params $sigs]
set bti [java::cast com.ibm.websphere.scheduler.BeanTaskInfo  $ti]
puts "Created the BeanTaskInfo object: $bti"
```

Important: Creating a BeanTaskInfo object does not add the task to the persistent store. Rather, it creates a placeholder for the necessary data. The task is not added to the persistent store until the create() method is called on a Scheduler, as described in the Submitting tasks to schedulers topic.

4. Set parameters on the BeanTaskInfo object. These parameters define which session bean is called and when. The TaskInfo interface contains various set() methods that you can use to control execution of the task, including when the task runs and what work the task does when it runs.

   The BeanTaskInfo interface requires that the TaskHandler Java™ Naming and Directory Interface (JNDI) name or TaskHandlerHome is set using the setTaskHandler method. If using the WASScheduler MBean API to set the task handler, then the JNDI name must be the fully-qualified global JNDI name.

   The TaskInfo interface specifies additional control points, as documented in the API documentation. Set parameters using the TaskInfo interface API method as shown in the following code example:

```
//create a date object which represents 30 seconds from now
java.util.Date startDate = new java.util.Date(System.currentTimeMillis()+30000);

//find the session bean to be called when the task starts
Object o = new InitialContext().lookup("java:comp/env/ejb/MyTaskHandlerHome");
TaskHandlerHome home = (TaskHandlerHome)javax.rmi.PortableRemoteObject.narrow(o,TaskHandlerHome.class);

//now set the start time and task handler to be called in the task info
taskInfo.setTaskHandler(home);
taskInfo.setStartTime(startDate);
```

   You can also set parameters using the following JACL scripting example:

```
# Setup the task
puts "Setting up the task..."
# Set the startTime if you want the task to run at a specific time, for example:
$bti setStartTime [java::new {java.util.Date long} [java::call System currentTimeMillis]]

# Set the StartTimeInterval so the task runs in 30 seconds from now
$bti setStartTimeInterval 30seconds

# Set JNDI name of the EJB which will get called when the task runs.  Since there is no
# application J2EE Context when the task is created by the MBean, this must be a
# global JNDI name.
$bti setTaskHandler $taskHandlerHomeJNDIName

# Do not purge the task when it's complete
$bti setAutoPurge false

# Set the name of the task.  This can be any string value.
$bti setName Created_by_MBean

# If the task needs to run with specific authorization you can set the tasks Authentication Alias
# Authentication aliases are created using the Admin Console.
# $bti setAuthenticationAlias {myRealm/myAlias}

puts "Task setup completed."
```

Results

A BeanTaskInfo object has been created that contains all of the relevant data to call an EJB method.

What to do next

Submit the task to a scheduler for creation.

Developing a task that sends a Java Message Service message

The Scheduler API and WASScheduler MBean API support different implementations of the TaskInfo interface, each of which can be used to schedule a particular type of work. This topic describes how to create a task that sends a Java Message Service (JMS) message to a queue or topic.

About this task

To create a task that sends a Java Message Service (JMS) message to a queue or topic, use these steps.

Procedure

1. Create an instance of the MessageTaskInfo interface using the Scheduler.createTaskInfo() factory method. Using a JavaServer Pages (JSP) file, servlet or EJB container, create the instance as shown in the following code example:

```
//lookup the scheduler to be used
Scheduler scheduler = (Scheduler)new InitialContext.lookup("java:comp/env/Scheduler");

MessageTaskInfo taskInfo = (MessageTaskInfo) scheduler.createTaskInfo(MessageTaskInfo.class);
```

   You can also use the wsadmin tool, create the instance as shown in the following JACL scripting example:

```
# Sample create a task using MessageTaskInfo task type
# Call this mbean with the following parameters:
#    <scheduler jndiName>     = JNDI name of the scheduler resource,
#                               for example scheduler/myScheduler
#    <JNDI name of the QCF>   = The global JNDI name of the Queue Connection Factory.
#    <JNDI name of the Queue> = The global JNDI name of the Queue destination

set jndiName [lindex $argv 0]
set jndiName_QCF [lindex $argv 1]
set jndiName_Q [lindex $argv 2]

# Map the JNDI name to the mbean name.  The mbean name is formed by replacing the / in the jndi name
# with . and prepending Scheduler_
regsub -all {/} $jndiName "." jndiName
set mbeanName Scheduler_$jndiName

puts "Looking-up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
puts $sched

# Get the ObjectName format of the Scheduler MBean
set schedO [$AdminControl makeObjectName $sched]

# Create a MessageTaskInfo object using invoke_jmx
puts "Creating MessageTaskInfo"
set params [java::new {java.lang.Object[]} 1]
$params set 0 [java::field com.ibm.websphere.scheduler.MessageTaskInfo class]

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 java.lang.Class

set ti [$AdminControl invoke_jmx $schedO createTaskInfo $params $sigs]
set mti [java::cast com.ibm.websphere.scheduler.MessageTaskInfo  $ti]
puts "Created the MessageTaskInfo object: $mti"
```

   Attention: Creating a MessageTaskInfo object does not add the task to the persistent store. Rather, it creates a placeholder for the necessary data. The task is not added to the persistent store until the create() method is called on a Scheduler, as described in the Submitting a task to a scheduler topic.

2. Set parameters on the MessageTaskInfo object. The TaskInfo interface contains various set() methods that can be used to control execution of the task, including when the task runs and what work the task does when it starts.

The TaskInfo interface specifies additional behavior settings, as documented in the API documentation. Using a JavaServer Pages (JSP) file, servlet or EJB container, create the instance as shown in the following code example:

```
//create a date object which represents 30 seconds from now
java.util.Date startDate = new java.util.Date(System.currentTimeMillis()+30000);


//now set the start time and the JNDI names for the queue connection factory and the queue
taskInfo.setConnectionFactoryJndiName("jms/MyQueueConnectionFactory");
taskInfo.setDestination("jms/MyQueue");
taskInfo.setStartTime(startDate);
```

You can also use the wsadmin tool, to create the instance as shown in the following JACL scripting example:

```
# Setup the task
puts "Setting up the task..."
# Set the startTime if you want the task to run at a specific time, for example:
$mti setStartTime [java::new {java.util.Date long} [java::call System currentTimeMillis]]

# Set the StartTimeInterval so the task runs in 30 seconds from now
$mti setStartTimeInterval 30seconds

# Set the global JNDI name of the QCF & Queue to send the message to.
$mti setConnectionFactoryJndiName $jndiName_QCF
$mti setDestinationJndiName $jndiName_Q

# Set the message
$mti setMessageData "Test Message"

# Do not purge the task when it's complete
$mti setAutoPurge false

# Set the name of the task.  This can be any string value.
$mti setName Created_by_MBean

# If the task needs to run with specific authorization you can set the tasks Authentication Alias
# Authentication aliases are created using the Admin Console.
# $mti setAuthenticationAlias {myRealm/myAlias}

puts "Task setup completed."
```

Results

A MessageTaskInfo object has been created that contains all of the relevant data for a task that sends a JMS message.

What to do next

Submit the task to a scheduler for creation, as described in the Submitting a task to a scheduler topic.

Scheduling long-running tasks

The default behavior of the scheduler is designed to run business logic that runs for a short period of time. In version 6.0.2 and later, two API methods on the com.ibm.websphere.scheduler.TaskInfo interface help avoid some of the problems that can occur when running tasks for an extended time.

About this task

The TaskInfo.setQOS method supports tasks with both a transactional and non-transactional quality of service. When running tasks that run for long periods, you can use the TaskInfo.QOS_ATLEASTONCE quality of service to run the task without a global transaction. This process prevents various timeout issues

that can occur when resources are held by a long-running transaction. See the Transactions and schedulers topic for details on the TaskInfo.setQOS method and how it can be used.

Using the TaskInfo.setExpectedDuration method, the scheduler can to adjust timeout values, as appropriate, for a given task for all qualities of service. The application server attempts to adjust various run-time parameters to accommodate the estimated run time of the task.

Procedure

1. When you assemble the TaskInfo object with the Scheduler API or the WASScheduler MBean, use the following methods on the TaskInfo interface:
   a. Set the quality of service.
      1) If the task must be transactional, use the setQOS method with the QOS_ONLYONCE constant, which is the default, if not set.
      2) If the task does not need to be transactional, use the setQOS method with the QOS_ATLEASTONCE constant.
   b. Set the expected duration.
      1) Use the setExpectedDuration method to set the expected duration of the task in seconds.
2. Schedule the task using the Scheduler.create method.

What to do next

View the Access schedulers topic.

Receiving scheduler notifications

Various notification events are generated by a scheduler when it performs an operation on a task. These notifications events are described in this topic.

About this task

The notification events generated by a scheduler when it performs a task include:

Scheduled
A task has been scheduled.

Purged
A task has been permanently deleted from the persistent store.

Suspended
A task was suspended.

Resumed
A task was resumed.

Complete
A task has run completely. If it was a repeating task, all repeats have been performed.

Cancelled
A task has been cancelled. It will not run again.

Firing A task is prepared to run.

Fired A task completed successfully.

Fire failed
A task could not run successfully.

To receive notification events, call the setNotificationSink() method on the TaskInfo interface before creating the task. The setNotificationSink() method enables you to specify the session bean that is to act as the callback, and a mask that restricts which events are generated.

Procedure

1. Create a NotificationSink session bean. Create a stateless session bean that implements the handleEvent() method in the com.ibm.websphere.scheduler.NotificationSink remote interface. The handleEvent() method is called when the notification is fired. The Home and Remote interfaces can be set as follows in the bean's deployment descriptor:

   ```
   com.ibm.websphere.scheduler.NotificationSinkHome
   com.ibm.websphere.scheduler.NotificationSink
   ```

   The NotificationSink interface defines the following method:

   ```
   public void handleEvent(TaskNotificationInfo task) throws java.rmi.RemoteException;
   ```

2. Specify the notification sink session bean prior to submitting the task to the Scheduler using the TaskInfo interface API setNotificationSink() method.

   If using the WASScheduler MBean API to set the notification sink, then the Java™ Naming and Directory Interface (JNDI) name must be the fully-qualified global JNDI name. Using a JavaServer Pages (JSP) file, servlet or Enterprise JavaBeans (EJB) component, look up and set the notification sink on a task as shown in the following code example:

   ```
   TaskInfo taskInfo = ...
   Object o = new InitialContext().lookup("java:comp/env/ejb/NotificationSink");
   NotificationSinkHome home = (NotificationSinkHome )javax.rmi.PortableRemoteObject.narrow
   (o,NotificationSinkHome.class);
   taskInfo.setNotificationSink(home,TaskNotificationInfo.ALL_EVENTS);
   ```

   You can also use the wsadmin tool to set the notification sink callback session bean as shown in the following JACL scripting example:

   ```
   # Use the NotificationSinkHome's Global JNDI name
   # Assume that a TaskInfo was already created...
   $taskInfo setNotificationSink "ejb/MyNotificationSink"
   ```

3. Specify the event mask. The event mask is specified as an integer bitmap. You can either use an individual mask such as TaskNotificationInfo.CREATED to receive specific events, TaskNotificationInfo.ALL_EVENTS to receive all events or a combination of specific events. If you use Java, your script might look like the following example:

   ```
   int eventMask = TaskNotificationInfo.FIRED | TaskNotificationInfo.COMPLETE;
   taskInfo.setNotificationSink(home,eventMask);
   ```

   If you use JACL, your script might look like the following example:

   ```
   # Set the event mask based on two event constants.
   set eventmask [expr [java::field com.ibm.websphere.scheduler.TaskNotificationInfo FIRED] +
    [java::field com.ibm.websphere.scheduler.TaskNotificationInfo COMPLETE]]

   # Set our Notification Sink based on our global JNDI name AND event mask.
   # Note:  We need to use the full method signature here since the
   # method resolver can't always detect the right method.
   $taskInfo {setNotificationSink String int} "ejb/MyNotificationSink" $eventmask
   ```

Results

A notification sink bean is now set on a TaskInfo object and can now be submitted to a scheduler using the create method.

Submitting a task to a scheduler

This topic describes the process of submitting a task to a configured scheduler.

Before you begin

This task assumes that you have already configured a scheduler and created and configured a TaskInfo object that calls a session bean or sends a Java Messaging Service (JMS) message.

About this task

Once you have developed a TaskInfo object that contains all relevant data for a task, submit the task to a scheduler for creation. When the task is created, the scheduler runs it.

Procedure

Create the task. After you configure TaskInfo, submit it to the appropriate scheduler, using the Scheduler API create method.

```
// Create the TaskInfo using the Scheduler that you already looked up and print out the Task ID
TaskStatus ts = scheduler.create(taskInfo);
System.out.println("Task created with id: " + ts.getTaskId()"
```

You can also create the task using the wsadmin tool as shown in the following JACL scripting example:

```
# Create the TaskInfo using the WASScheduler MBean that you previously located and print out the Task ID
puts "Creating the task..."

set params [java::new {java.lang.Object[]} 1]
$params set 0 $taskInfo

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 com.ibm.websphere.scheduler.TaskInfo

set taskStatus [java::cast com.ibm.websphere.scheduler.TaskStatus [$AdminControl invoke_jmx $schedO
 create $params $sigs]]

puts "Task Created.  TaskID= [$taskStatus getTaskId]"

puts $taskStatus
```

When the call to the create() method is complete, the task exists in the persistent store and is run at the time specified in the TaskInfo object. If a global transactional context is present on the thread, and the create() transaction rolls back or is aborted, the task does not run.
The TaskStatus object, which has been returned by the call to the create() method, contains information about the state of the task, as well as the task ID. The task ID is the unique identifier for this task, and is required if the task is to be suspended, resumed, cancelled, and so on, at a later time.

Tip: The TaskStatus object is only a snapshot of the current state of the task. Use the Scheduler.getStatus() method to receive the current state when needed.

Task management methods using a scheduler

The scheduler provides several task management methods.

When a task is created by calling the create() method on a scheduler, a TaskStatus object is returned to the caller. The TaskStatus object contains the task ID, which is a unique identifier. The Scheduler API and WASScheduler MBean define several additional methods that pertain to the management of tasks, each of which accepts the task ID as a parameter. The following task management methods are defined:

suspend()
: Suspends a task. The task does not run until it has been resumed.

resume()
: Resumes a previously suspended task.

cancel()
: Cancels a task. The task is not run and cannot be resumed.

purge()
: Permanently deletes a cancelled task from the persistent store.

getStatus()
: Returns the current status of the task.

Use the following API example to create and cancel a task:

```
//Create the task.
TaskInfo taskInfo = ...
TaskStatus status = scheduler.create(taskInfo);

//Get the task ID
String taskId = status.getTaskId();

//Cancel the task. Specify the purgeAlso flag so that the task does not remain in the persistent store
scheduler.cancel(taskId,true);
```

Use the following example JACL script operations in the wsadmin tool to create and cancel a task:

```
set jndiName sched/MyScheduler

# Map the JNDI name to the mbean name.  The mbean name is
# formed by replacing the / in the jndi name with . and prepending
# Scheduler_
regsub -all {/} $jndiName "." jndiName
set mbeanName Scheduler_$jndiName

puts "Looking-up Scheduler MBean $mbeanName"
set sched [$AdminControl queryNames WebSphere:*,type=WASScheduler,name=$mbeanName]
puts $sched

# Get the ObjectName format of the Scheduler MBean
set schedO [$AdminControl makeObjectName $sched]

# Create a TaskInfo object...
# (Some code excluded...)
set params [java::new {java.lang.Object[]} 1]
$params set 0 $taskInfo

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 com.ibm.websphere.scheduler.TaskInfo

set taskStatus [java::cast com.ibm.websphere.scheduler.TaskStatus [$AdminControl invoke_jmx $schedO
 create $params $sigs]]

set taskID [$taskStatus getTaskId]
puts "Task Created.  TaskID= $taskID"

# Cancel the task using the Task ID from the TaskStatus object returned during create.
set params [java::new {java.lang.Object[]} 1]
$params set 0 false

set sigs [java::new {java.lang.String[]} 1]
$sigs set 0 java.lang.boolean

set taskStatus [java::cast com.ibm.websphere.scheduler.TaskStatus [$AdminControl invoke_jmx $schedO
 cancel $params $sigs]]
```

Transactionality. All methods of the Scheduler API are transactional. If a global transactional context is present, it is used to perform the operation. If an unexpected exception is thrown, the transaction is marked to roll back, and the caller must handle it appropriately. If an expected or declared exception is thrown, the transaction remains intact and the caller must choose to roll back or to commit the transaction. If the transaction is rolled back at some point, all scheduler operations performed within the transaction are also rolled back.

If a local transactional context is present, it is suspended and a new global transactional context begins. Likewise, if no transactional context is active, a global transactional context begins. In both cases, if an unexpected exception is thrown, the transaction rolls back. If a declared exception is thrown, the transaction is committed.

If another thread is concurrently modifying the task in question, a TaskPending exception is thrown. This is because schedulers lock the database optimistically. The calling application can then retry the operation.

Task management functions may block if the task is currently running. Because the scheduler guarantees that each task will run only once, the task must be locked for the duration of a running task. Likewise, if a task is changed using one of the management functions but the global transaction is not committed, any other management functions issued from another transaction for that task will be blocked.

A stateless session bean task's TaskHandler.process() method can change it's own state. However, the task must be running within the same transaction as the scheduler. Therefore, a running task can only modify itself if it is using the Required or Mandatory container managed transaction types. If the Requires New transaction type is specified on the process() method, all management functions will deadlock.

All methods defined by the Scheduler API are described in the API documentation.

Identifying tasks that are currently running

When a task runs, the task database record is locked until the task completes. This topic describes how to determine whether or not a task is running.

About this task

Prior to version 6.0.2, all tasks ran in a single global transaction. This process not only prevented the task from running more than once successfully, but it also blocked all attempts at reading the state of the task, since each task used read-committed transaction isolation.

There are two methods for determining whether a task is running:

1. **NotificationSink**

   A NotificationSink EJB can be set on the task using the setNotificationSink method on the TaskInfo object. The NotificationSink bean can then log the life cycle of the task to a separate database record in a custom table. This would result in a history log of the task that can be queried independently from the scheduler. This solution works for all versions of the scheduler service. See the Receiving Scheduler Notifications topic for details.
2. **Delayed Execution and Uncommitted Read**

   In Version 6.0.2 and later, two behaviors enable the scheduler find and retrieve API methods, such as getTask, getTaskStatus or findTasksByName, to see the current state of the task without blocking. To see the current state of the task, including its uncommitted running state, complete the following steps:

Procedure

1. Enable read-uncommitted transaction isolation for the scheduler read methods to prevent these methods from blocking while a task is running. To set the default transaction isolation for read methods, see the Configuring scheduler default transaction isolation topic for read operation details.

   Important: If the scheduler database does not support uncommitted reads, such as Oracle, it might not be possible to determine if a task is running unless you use the QOS_ATLEASTONCE quality of service.
2. Use the TaskInfo.EXECUTION_DELAYEDUPDATE option on the TaskInfo.setTaskExecutionOptions method to force the scheduler to write the TaskStatus.RUNNING state to the task when that task starts running.

Stopping tasks that are failing

The scheduler runs tasks in a global transactional context, by default. If a task is failing due to a configuration problem or application error, the scheduler attempts to retry the task until the scheduler failure threshold is reached. This topic describes how to stop the tasks that are failing.

Before you begin

The default scheduler failure threshold is `10` and can be configured using the taskFailureThreshold scheduler custom property. To configure this custom property, in the administrative console click **Resources** > **Schedulers** > ***scheduler_name*** and click **Custom Properties**. Valid values are non-negative integers, that is, integers greater than `0`.

About this task

When the task reaches the failure threshold, the scheduler stops running the task until the scheduler daemon is restarted using the WASScheduler MBean, the scheduler fails over to another server, or until the scheduler is resumed using the resume method on the Scheduler API or WASScheduler MBean.

Procedure

1. Cancel or suspend a transactional (QOS_ONLYONCE) task that is continually failing. This action can be difficult if the scheduler has not yet reached the failure threshold. The cancel and suspend Scheduler API methods or WASScheduler MBean operations block until the task fails or the method times out, while waiting for a database lock and throws a TaskPending exception. If this occurs, then the application can retry the cancel or suspend operation until it completes.
2. Alternatively, stop the scheduler daemon using the stopDaemon operation on the WASScheduler MBean to avoid running the task multiple times, and run the cancel or suspend operation while it is stopped. While the daemon is stopped, the scheduler does not run tasks. However, all MBean operations and API methods are still available.

Scheduler tasks and Java EE context

When a task is created using the Scheduler API create() method, the Java Platform, Enterprise Edition (Java EE) thread context of the creator is stored with the scheduled task. When the task runs, the original Java EE thread context is reapplied to the thread before calling the customer TaskInfo instance.

The scheduler service utilizes the asynchronous beans deferred start mechanism to propagate Java EE service context information to a task when it runs. The amount of service context information that is propagated is controlled by the Service Context settings on the WorkManager configuration object that schedulers reference. For example, security and internationalization service contexts can be enabled. See the Using asynchronous beans topic for details on how to configure the Application Server to propagate these service contexts.

Transactions and schedulers

The scheduler runs a task in a single global transaction, by default. You can use the QOS_ONLYONCE or QOS_ATLEASTONCE quality of service to specify whether the task runs as a single unit of work once or as independent transactions.

Transaction behavior when running a task

Because the scheduler runs a task in a single global transaction, by default, the transaction is open until the task completes or fails. The resources involved in that transaction are subject to various timeouts and the thread of the task could be identified as hung if the task runs for a long period of time that can span many minutes or hours.

QOS_ONLYONCE

Scheduled tasks execute only one time successfully when using the QOS_ONLYONCE quality of service. This action is accomplished by grouping all of the work done in the task as a single unit of work. When each task fires, the following events occur in a single global transactional context:

1. The context of the application that created the task is applied to the thread.
2. A global transactional context is started.

3. The next fire time and start-by time are calculated using the UserCalendar bean or the DefaultUserCalendar.

   Important: If using the TaskInfo.setTaskExcecutionOptions method with the TaskInfo.EXECUTION_DELAYEDUPDATE option, this step will occur after the record is updated.
4. The task database task record is updated in the database with the state of the next task or deleted if the task is complete and the task's auto-purge setting is true.
5. The task database record is updated in the database with the state of the next task or deleted if the task is complete and the task's auto-purge setting is true. If the EXECUTION_DELAYEDUPDATE option is used, the database will not reflect the next state of the task, but the current state with the TaskStatus.RUNNING state set.
6. If the NotificationSink bean is set, a FIRING notification is fired.
7. The BeanTaskInfo or MessageTaskInfo object starts.
8. If the task fails and the NotificationSink bean is set, a FIRE_FAILED notification is fired on a separate transaction.
9. If the task's NotificationSink bean is set, then the various notifications are fired as required.
10. If the EXECUTION_DELAYEDUPDATE option is used for the task, the database will be updated a second time with the next state of the task.
11. The global transaction is committed.

Because all events belonging to a task are executed in a single global transactional context, consider the following points in order to avoid transaction-related errors:

- Each resource participating in the task transaction must be two-phase XA capable.

  This includes the Java Database Connectivity (JDBC) datasource that is configured for the scheduler, any Java Messaging Service (JMS) services used by the MessageTaskInfo objects, and any resources used within any of the UserCalendar, TaskHandler, or NotificationSink beans that have a transaction setting of "Required".
- One resource can be single-phase, if last participant support is enabled for the application that created the transaction. Enable last participant support using an assembly tool. You can also enable last participant support through the administrative console. See the Last participant support extension settings topic for details.

All unexpected exceptions are logged to the activity log and all events participating in the task global transaction are rolled back. This includes changes to the task database record, which force the task to be executed again when the scheduler daemon polls the database during the next poll cycle. The UserCalendar, TaskHandler, and NotificationSink beans can choose not to participate in the global transaction by configuring the bean transaction setting to "Requires new".

QOS_ATLEASTONCE

Scheduled tasks that use the QOS_ATLEASTONCE quality of service do not have a single transactional context. In this case, each calendar calculation, event notification and database update occurs in an independent transaction:

1. The context of the application that created the task is applied to the thread.
2. The task’s database record is updated with the RUNNING state of the task.
3. UserCalendar, NotificationSink beans are called.
4. The BeanTaskInfo or MessageTaskInfo is started.
5. Result notifications are sent.
6. The database is updated with the next state of the task, if the task has not been changed since the RUNNING state was written.

If a failure happens after the RUNNING state is written to the database and before the result is written, then the task may run more than one time.

When using QOS_ATLEASTONCE, all NotificationSink, UserCalendar and TaskHandler beans must not mandate a transaction (TX_MANDATORY), since there is no global transaction available when the task runs. The EJB components use "Required" or "Requires new" container managed transaction or a bean managed transaction.

Transaction behavior when using the Scheduler API methods or WASScheduler MBean operations

All Scheduler interface methods participate in a single global transactional context. If a global transactional context is already present on the thread when the create(), suspend(), resume(), cancel(), and purge() methods are executed, then the existing global transaction is used. Otherwise, a new global transaction begins.

If the method participates in the global transaction of the caller and an unexpected error occurs, then the transaction is marked to roll back. If the exception is a declared exception, then the exception is resubmitted to the caller, and the transaction is left alone for the caller to commit or roll back.

If the method starts its own global transaction and any exception occurs, then the transaction is rolled back, and the exception is resubmitted to the caller.

Scheduler task user authorization

The scheduler service uses the asynchronous beans deferred start mechanism to propagate Java Platform, Enterprise Edition (Java EE) service context information to a task when it runs. If you plan to secure your application using the JAAS security context of the administrative security mechanism built into WebSphere Application Server, create each task with the correct credentials on the thread.

Tasks run with specified security credentials using the following methods:

- Using the Java Authentication and Authorization Service (JAAS) security context on the thread at the time the task was created. See the Deferred start and security topic in the Asynchronous beans section of the information center.
- Using the setAuthenticationAlias method on the TaskInfo object.
- Using a specified security identity on a BeanTaskInfo task TaskHandler EJB method.

The scheduler service utilizes the asynchronous beans deferred start mechanism to propagate Java EE service context information to a task when it runs. The amount of service context information that is propagated is controlled by the Service Context settings on the WorkManager configuration object that schedulers reference. For example, security and internationalization service contexts can be enabled. See the Using asynchronous beans topic for details on how to configure the Application Server to propagate these service contexts.

Java Authentication and Authorization Service Security context

If you intend to secure your application using the JAAS security context of the administrative security mechanism built into WebSphere Application Server, create each task with the correct credentials on the thread. Once each task has the correct credentials, you can disable and re-enable administrative security without causing any security problems. If you do not set the security context when the scheduler task is created and you later enable security in the target application, a security exception or error message might display, such as SECJ0053E. You might also see this error if two or more schedulers on different servers are accessing the same tables (a clustered or redundant scheduler) and the security settings are different.

The JAAS security context is not set if any of the follow conditions are true:

- Administrative security is disabled.
- Security context policies are disabled on the configured WorkManager for the associated scheduler configuration.

- A credential is not set on the thread. For example, the enterprise bean or servlet that is used to create the scheduled task is not secured, or the task was created with a WASScheduler MBean.

If any of the previously mentioned conditions are true when you create your task and you need to enable security on your application server or application, you must complete the following steps for each task:

1. Find the task using the Scheduler API find or get methods.
2. Cancel the task using the Scheduler.cancel() API.
3. Recreate the task using the Scheduler.create() method with security enabled. Submitting a task that was retrieved from the scheduler using the find or get methods will automatically generate a new task ID.

Security order of precedence

As previously noted, there are three ways of verifying that a task will run with the correct user credentials. In addition, each TaskInfo implementation may have its own way of supplying user information, which may override the standard mechanisms. If multiple methods are used, refer to the following lists to determine which security mechanism is going to be employed.

BeanTaskInfo

1. TaskHandler security identity set on the process() method of the Enterprise Java Bean file
2. Authentication Alias set with the setAuthenticationAlias method on the TaskInfo interface
3. JAAS security context

MessageTaskInfo

1. Authentication Alias set with the setAuthenticationAlias method on the TaskInfo interface
2. The setUsername and setPassword methods on the MessageTaskInfo interface. See the Deprecated features article for more information.

Securing scheduler tasks

Scheduled tasks are protected using application isolation and administrative roles. This topic describes how to secure scheduler tasks.

About this task

If a task is created using a Java Platform, Enterprise Edition (Java EE) server application, only applications with the same name can access those tasks. Tasks created with a WASScheduler MBean using the AdminClient interface or scripting are not part of a J2EE application and have access to all tasks regardless of the application with which they were created. Tasks created with a WASScheduler MBean are only accessible from the WASScheduler MBean API and are not accessible from the Scheduler API.

If the Use Administration Roles attribute is enabled on a scheduler and administrative security is enabled on the Application Server, all Scheduler API methods and WASScheduler MBean API operations enforce access based on the WebSphere Administration Roles. If either of these attributes are disabled, then all API methods are fully accessible by all users.

Procedure

1. Enable security for all application servers.
2. Manage schedulers.

Scheduler configuration or topology

The scheduler uses a database to persist information concerning which tasks to run and when. Errors might occur when changing the application server topology or when changing the application or server configuration. When you change the configuration or topology, carefully consider how this action affects the scheduler.

Restricting security

If you created tasks with an application server while security is disabled, and you later decide to enable security, then the scheduler might have difficulty running tasks. When you create a task, the security context of the application thread is automatically stored with the task. If security is not stored with the task (see Scheduler task user authorization), and you later enable security on the server or application where the task is to run, then the following errors might be logged:

```
SECJ0053E: Authorization failed for /UNAUTHENTICATED while invoking (Home)com/ibm/websphere/scheduler
/TaskHandler create:2 securityName: /UNAUTHENTICATED;accessID: UNAUTHENTICATED is not granted any of
the required roles: MySecurityRole
```

Before you enable security on the server or application, determine if any tasks might be adversely affected. If so, use the Scheduler API or WASScheduler MBean to cancel the tasks and recreate them after you configure security.

Application server topology changes

The scheduler stores javax.ejb.HomeHandle objects for TaskHandler, NotificationSink and UserCalendar *homes* when the task is created. When you run the task later, these home handles are reinflated and used to access the Enterprise JavaBeans (EJB) component home. When the home handle references an EJB on a single-server environment, the home handles have affinity to that server. When the home handle references an EJB component on a cluster, then the home handles have affinity to the cluster.

If the application server or the Workload Managed (WLM) cluster that a home handle is referencing is not available, then the scheduler fails to run the task, and the following error is logged:

```
SCHD0063E: A task with ID 123 failed to run on Scheduler MyScheduler (sched/MyScheduler) because of
an exception: {cause of failure}
```

If you upgrade the application server to a cluster, or if the Object Request Broker (ORB) ORB_LISTENER_ADDRESS is not set to a fixed port number (see Configuring Inbound Transports), then the task might also fail, since the information stored within the home handle does not have the appropriate information to find the desired server.

Upgrading to a scheduler cluster

A scheduler cluster (not to be confused with a WLM cluster) is a collection of scheduler configurations on different application servers that share the same Java Naming and Directory Interface (JNDI) name, Java Database Connectivity (JDBC) data source and table prefix. If you upgrade a stand-alone scheduler to a clustered scheduler, then the application and any associated resources that the application requires must be available. If this is not the case, the scheduled task fails to run and error messages might be logged:

```
SCHD0103W: The Scheduler MyScheduler (sched/MyScheduler) was unable to run task 123 because the
application or module is unavailable: MyTaskHandlerEJB
```

To avoid issues with application availability and achieve optimal results, use the same servers in a scheduler cluster as those used in a WLM cluster.

Reusing scheduler tables

When changing any topology, moving from development to production environments, or making any configuration changes that make the environment more restrictive, you might get optimal results if you use

a different set of scheduler tables. Reusing scheduler tables that have scheduled tasks from previous releases without careful planning might cause problems:

- EJB components running on unexpected application servers.
- Tasks failing to run due to invalid or missing security credentials.
- Tasks failing to run due to invalid or missing Java Platform, Enterprise Edition (Java EE) context information.

Diagnosing such problems is challenging and requires analyzing logs on all servers that have a scheduler installed and configured. When the problem tasks are located, the tasks can be cancelled using the Scheduler APIs, or the tables can be dropped and recreated.

Scheduler interface

Use the `com.ibm.websphere.scheduler.Scheduler` Java object (in the Java™ Naming and Directory Interface (JNDI) namespace for the scheduler configuration) to find a reference to a scheduler and work with tasks.

A `com.ibm.websphere.scheduler.Scheduler` Java object exists in the JNDI namespace for each scheduler configuration. A reference to a scheduler can be obtained by performing a lookup on the JNDI name; however, the lookup is valid only from the server process where the scheduler instance exists. Once a reference has been obtained, tasks can be created, suspended, cancelled, and so on, if the caller has access to the scheduler instance.

For details, see Interface Scheduler in the API documentation.

Task creation
: The task is created in the persistent store using the global transactional context of the caller, if present. See the topic, “Transactions and schedulers” on page 868, for more details. Since this is a transactional operation, the task cannot be run or modified from another thread until the current transaction commits.

Task modification
: Tasks that have been created can be modified with the suspend(), resume(), cancel(), and purge() methods. These methods take a Task Identifier string as a parameter, which is generated by the create() method and can be found in the TaskStatus object. If a task is currently running or being modified by another thread, an operation that attempts to modify the state of the task might block on the attempt. Tasks can only be modified by the same application (EAR file) that was used to create the task.

Task execution
: Tasks are run in the thread pool specified by the configuration's work manager. If multiple schedulers are configured to share the same database tables, the scheduler is clustered and the tasks found in the table can be run on any of the schedulers, whether or not they are in the same server, node, or cell.

Task lookup
: Tasks can be located using the Name property that was assigned at creation time. This is useful when you need to modify a group of tasks and tracking individual task ID's is not convenient.

TaskInfo interface

TaskInfo objects contain the information to create a task. Several implementations of this class exist, one for each type of task that can run.

Available TaskInfo implementations include:

BeanTaskInfo
: Calls a stateless session bean.

MessageTaskInfo
: Sends a Java™ Messaging Service (JMS) message to a queue or publishes a message to a topic. For details, see the Interface TaskInfo in the API documentation.

After a TaskInfo object is created, it can be submitted to the scheduler for task creation by calling the Scheduler.create() method.

For details about the TaskInfo interface, see the API documentation .

TaskHandler interface

A task handler is a user-defined stateless session bean that is called by tasks created using a BeanTaskInfo object.

A task handler bean uses the following home and remote interfaces, which are defined in the deployment descriptor using an assembly tool, such as Rational Application Developer:

```
com.ibm.websphere.scheduler.TaskHandlerHome
com.ibm.websphere.scheduler.TaskHandler
```

The bean itself needs to implement the process() method defined in the remote interface. For details, see the Interface TaskHandler in the API documentation.

Once an EJB is created and available within an enterprise application, it can be called by a BeanTaskInfo task when it runs. See the Developing a task that calls a session bean topic for details.

When a task is created using a BeanTaskInfo object, the process() method on the TaskHandler session bean is called whenever the task runs. Because the TaskStatus object for the task is passed as a parameter to the process() method, the task handler determines different types of information about the task, such as when it will fire next, the number of repeats remaining, its name and its ID.

The process() method can also change its own state. However, the task must be running within the same transaction as the scheduler. Therefore, a running task can only modify itself if it is using the **Required** or **Mandatory** container managed transaction types. If the **Requires New** transaction type is specified on the process()method, all management functions deadlock.

NotificationSink interface

A notification sink is a user-defined stateless session bean that is called when the task changes state.

A notification sink bean uses the following home and remote interfaces, which are defined in the deployment descriptor using an assembly tool, such as Rational Application Developer:

```
com.ibm.websphere.scheduler.NotificationSinkHome
com.ibm.websphere.scheduler.NotificationSink
```

The bean itself needs to implement the handleEvent() method defined in the remote interface. For details, see the Interface NotificationSink section of the API documentation and the Receiving scheduler notifications topic.

A NotificationSink provides an event notification callback on a task-by-task basis. A notification sink is set on the TaskInfo interface, using the setNotificationSink() method. If a notification sink is not specified on a task, all notifications are lost; however, the status of a task can be determined by calling the getStatus() method from the Scheduler interface. A notification callback is made for each of the following events:

- Scheduled
- Suspended
- Resumed
- Fired
- Firing
- Fire Failed
- Complete
- Purged

UserCalendar interface

A user calendar is a user-defined stateless session bean that is called by tasks when they need to calculate date-related values.

A user calendar bean uses the following home and remote interfaces, which are defined in the deployment descriptor using an assembly tool, such as Rational Application Developer:

```
com.ibm.websphere.scheduler.UserCalendarHome
com.ibm.websphere.scheduler.UserCalendar
```

The bean itself needs to implement the applyDelta(), validate(), and getCalendarNames() methods defined in the remote interface. For details, see the Interface UserCalendar in the API documentation.

User calendars are used to calculate time intervals, such as the time between task runs. A user calendar takes a java.util.Date object, applies the interval string and returns the resulting java.util.Date.

User calendars are set with the setUserCalendar() method on the TaskInfo interface and called by the scheduler run-time code when a delta calculation is necessary.

The following methods on the TaskInfo interface specify delta strings that use the user calendar for calculation:

- setStartTimeInterval
- setStartByInterval
- setRepeatInterval

Default user calendar

If a user calendar has not been specified using the TaskInfo.setUserCalendar() method, a default user calendar is used. The default calendar allows for simple delta specifications, such as seconds, minutes, hours, days, and months. See the API documentation for details on the default calendar. The default user calendar also provides a CRON-like syntax for calculating absolute times versus time deltas.

Calendar identifiers

A single user calendar can contain logic for multiple calendars. A calendar specifier string determines which calendar is used. For example, a calendar bean might be implemented to recognize the interval *day*. However, the identifier also recognizes two calendar implementations: *standard* (for a standard calendar day) and *business* (for a business day).

Internationalization and time zones

Scheduler makes use of the java.util.Date class when storing and processing dates. Internally, this class saves the time as milliseconds since the Epoch, Greenwich Mean Time. Since the Date is not converted to local time until converted to a string, the scheduler respects the time zone where the date was created.

Writing user calendars

Because the user calendar is a stateless session bean, the same Java Platform, Enterprise Edition (Java EE) programming model available to other session beans is available to the user calendar as well.

Chapter 21. Developing security

Secure specific types of applications, such as applications that include portlets, SIP servlets, enterprise beans, web services. Find security information that focuses on specific concerns, such as messaging, transaction support, naming and directory, data access.

Developing extensions to the WebSphere security infrastructure

WebSphere Application Server provides various plug points so that you can extend the security infrastructure. Extending this security infrastructure involves several activities including: Developing custom user registries, developing applications that use programmatic security, and customizing web application login forms.

About this task

The following topics are covered in this section:

Procedure

- Developing custom user registries
- Developing applications that use programmatic security
- Customizing web application login forms
- Customizing application login forms with Java Authentication and Authorization Service (JAAS)
- Securing transports with Java Secure Sockets Extension (JSSE) and Java Cryptography Extension (JCE) programming interfaces
- Implementing tokens for security attribute propagation
- Implementing a custom authentication provider using JASPI

Developing stand-alone custom registries

This development provides considerable flexibility in adapting WebSphere Application Server security to various environments where some notion of a user registry, other than LDAP or Local OS, already exists in the operational environment.

Before you begin

WebSphere Application Server security supports the use of stand-alone custom registries in addition to the local operating system registry, stand-alone Lightweight Directory Access Protocol (LDAP) registries, and federated repositories for authentication and authorization purposes. A stand-alone custom-implemented registry uses the UserRegistry Java interface as provided by WebSphere Application Server. A stand-alone custom-implemented registry can support virtually any type or notion of an accounts repository from a relational database, flat file, and so on.

Implementing a stand-alone custom registry is a software development effort. Implement the methods that are defined in the com.ibm.websphere.security.UserRegistry interface to make calls to the appropriate registry to obtain user and group information. The interface defines a general set of methods for encapsulating a wide variety of registries. You can configure a stand-alone custom registry as the selected repository when configuring WebSphere Application Server security on the Global security panel.

In WebSphere Application Server Version 8.5, make sure that your implementation of the stand-alone custom registry does not depend on any WebSphere Application Server components such as data sources, Enterprise JavaBeans (EJB) and Java Naming and Directory Interface (JNDI). You can not have this dependency because security is initialized and enabled prior to most of the other WebSphere Application Server components during startup. If your previous implementation used these components,

make a change that eliminates the dependency. For example, if your previous implementation used data sources to connect to a database, instead use the JDBC java.sql.DriverManager interface to connect to the database.

Note: The user registry is used in controllers and servants. There is an increased risk of integrity exposure in that configuration if the registry implementation is not secured.

If your previous implementation uses data sources to connect to a database, change the implementation to use Java database connectivity (JDBC) connections.

Procedure

1. Implement all the methods in the interface except for the CreateCredential method, which is implemented by WebSphere Application Server.
2. Build your implementation.

   To compile your code, you need the *app_server_root*/plugins/com.ibm.ws.runtime.jar and the *app_server_root*/plugins/com.ibm.ws.security.crypto.jar files in your class path. For example:

   ```
   app_server_root\java\bin\javac -classpath
   app_server_root\plugins\com.ibm.ws.runtime.jar:
   app_server_root\plugins\com.ibm.ws.security.crypto.jar your_implementation_file.java
   ```
3. Copy the class files that are generated in the previous step to the product class path.

   The preferred location is the following directory:

   - *%install_root%*\lib\ext

   directory. Copy these class files to all of the product process class paths.
4. To configure your implementation using the administrative console, follow the steps in topics about configuring stand-alone custom registries. This step is required to implement custom user registries.

Example

Viewing stand-alone custom registries.

Use these links to view registry examples.

A *stand-alone custom registry* is a customer-implemented registry that implements the UserRegistry Java interface, as provided by WebSphere Application Server. A custom-implemented registry can support virtually any type or form of an accounts repository from a relational database, flat file, and so on. The custom registry provides considerable flexibility in adapting WebSphere Application Server security to various environments where some form of a registry, other than a federated repository, Lightweight Directory Access Protocol (LDAP) registry, or local operating system registry, already exist in the operational environment.

What to do next

If you enable security, make sure that you complete the remaining steps:

1. Save and synchronize the configuration and restart all of the servers.
2. Try accessing some J2EE resources to verify that the custom registry implementation is correct.

Result.java file

This module is used by user registries in WebSphere Application Server when calling the getUsers and getGroups methods. The user registries use this method to set the list of users and groups and to indicate if more users and groups in the user registry exist than requested.

```
//
//  5639-D57, 5630-A36, 5630-A37, 5724-D18
// (C) COPYRIGHT International Business Machines Corp. 1997, 2005
//  All Rights Reserved * Licensed Materials - Property of IBM
//
```

```
package com.ibm.websphere.security;

import java.util.List;

public class Result implements java.io.Serializable {
    /**
      Default constructor
    */
    public Result() {
    }

    /**
       Returns the list of users and groups
       @return the list of users and groups
    */
    public List getList() {
      return list;
    }

    /**
       indicates if there are more users and groups in the registry
    */
    public boolean hasMore() {
      return more;
    }
    /**
       Set the flag to indicate that there are more users and groups
       in the registry to true
    */
    public void setHasMore() {
      more = true;
    }

    /*
      Set the list of users and groups
      @param list   list of users/groups
    */
    public void setList(List list) {
      this.list = list;
    }

    private boolean more = false;
    private List list;
}
```

UserRegistry.java files

The following file is a custom property that is used with a custom user registry.

For more information, see Configuring stand-alone custom registries.

```
// 5639-D57, 5630-A36, 5630-A37, 5724-D18
// (C) COPYRIGHT International Business Machines Corp. 1997, 2005
// All Rights Reserved * Licensed Materials - Property of IBM
//
// DESCRIPTION:
//
//    This file is the UserRegistry interface that custom registries in WebSphere
//    Application Server implement to enable WebSphere security to use the custom
//    registry.
//

package com.ibm.websphere.security;

import java.util.*;
import java.rmi.*;
import java.security.cert.X509Certificate;
import com.ibm.websphere.security.cred.WSCredential;

/**
 * Implementing this interface enables WebSphere Application Server Security
 * to use custom registries. This interface extends java.rmi.Remote because the
 * registry can be in a remote process.
 *
 * Implementation of this interface must provide implementations for:
*
* initialize(java.util.Properties)
* checkPassword(String,String)
* mapCertificate(X509Certificate[])
* getRealm
* getUsers(String,int)
* getUserDisplayName(String)
* getUniqueUserId(String)
* getUserSecurityName(String)
* isValidUser(String)
* getGroups(String,int)
* getGroupDisplayName(String)
* getUniqueGroupId(String)
* getUniqueGroupIds(String)
```

```
* getGroupSecurityName(String)
* isValidGroup(String)
* getGroupsForUser(String)
* getUsersForGroup(String,int)
* createCredential(String)
**/

public interface UserRegistry extends java.rmi.Remote
{

  /**
   * Initializes the registry. This method is called when creating the
   * registry.
   *
   * @param props the registry-specific properties with which to
   *               initialize the  custom registry
   * @exception CustomRegistryException
   *                    if there is any registry specific problem
   * @exception RemoteException
   *    as this extends java.rmi.Remote
   **/
   public void initialize(java.util.Properties props)
      throws CustomRegistryException,
             RemoteException;
  /**
   * Checks the password of the user. This method is called to authenticate a
   * user when the user's name and password are given.
   *
   * @param userSecurityName the name of the user
   * @param password the password of the user
   * @return a valid userSecurityName. Normally this is
   *   the name of same user whose password was checked but if the
   *  implementation wants to return any other valid
   *  userSecurityName in the registry it can do so
   * @exception CheckPasswordFailedException if userSecurityName/
   *  password combination does not exist in the registry
   * @exception CustomRegistryException if there is any registry specific
   *            problem
   * @exception RemoteException as this extends java.rmi.Remote
   **/
   public String checkPassword(String userSecurityName, String password)
      throws PasswordCheckFailedException,
             CustomRegistryException,
             RemoteException;

  /**
   * Maps a certificate (of X509 format) to a valid user in the registry.
   * This is used to map the name in the certificate supplied by a browser
   * to a valid userSecurityName in the registry
   *
   * @param cert the X509 certificate chain
   * @return the mapped name of the user userSecurityName
   * @exception CertificateMapNotSupportedException if the particular
   *            certificate is not supported.
   * @exception CertificateMapFailedException if the mapping of the
   *            certificate fails.
   * @exception CustomRegistryException if there is any registry specific
   *            problem
   * @exception RemoteException as this extends java.rmi.Remote
   **/
   public String mapCertificate(X509Certificate[] cert)
      throws CertificateMapNotSupportedException,
             CertificateMapFailedException,
             CustomRegistryException,
             RemoteException;

  /**
   * Returns the realm of the registry.
   *
   * @return the realm. The realm is a registry-specific string indicating
   *            the realm or domain for which this registry
   *            applies.  For example, for OS400 or AIX this would be the
   *            host name of the system whose user registry this object
   *            represents.
   *            If null is returned by this method realm defaults to the
   *            value of "customRealm". It is recommended that you use
   *            your own value for realm.
   * @exception CustomRegistryException if there is any registry specific
   *            problem
   * @exception RemoteException as this extends java.rmi.Remote
   **/
   public String getRealm()
      throws CustomRegistryException,
             RemoteException;

  /**
   * Gets a list of users that match a pattern in the registry.
   * The maximum number of users returned is defined by the limit
   * argument.
   * This method is called by administrative console and by scripting (command
```

```
 * line) to make available the users in the registry for adding them (users)
 * to roles.
 *
 * @parameter pattern the pattern to match. (For example., a* will match all
 *   userSecurityNames starting with a)
 * @parameter limit the maximum number of users that should be returned.
 *  This is very useful in situations where there are thousands of
 *            users in the registry and getting all of them at once is not
 *            practical. A value of 0 implies get all the users and hence
 *            must be used with care.
 * @return a Result object that contains the list of users
 *   requested and a flag to indicate if more users exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public Result getUsers(String pattern, int limit)
    throws CustomRegistryException,
           RemoteException;

/**
 * Returns the display name for the user specified by userSecurityName.
 *
 * This method is called only when the user information displays
 * (information purposes only, for example, in the administrative console) and not used
 * in the actual authentication or authorization purposes. If there are no
 * display names in the registry return null or empty string.
 *
 * In WebSphere Application Server Version 4.0 custom registry, if you had a display
 * name for the user and  if it was different from the security name, the display name
 * was returned for the EJB methods getCallerPrincipal() and the servlet methods
 * getUserPrincipal() and  getRemoteUser().
 * In WebSphere Application Server Version 6.0 for the same methods the security
 * name is returned by default. This is the recommended way as the display name
 * is not unique and might create security holes.
 *
 * See the documentation for more information.
 *
 * @parameter userSecurityName the name of the user.
 * @return the display name for the user. The display name
 *   is a registry-specific string that represents a descriptive, not
 *  necessarily unique, name for a user. If a display name does
 *            not exist return null or empty string.
 * @exception EntryNotFoundException if userSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUserDisplayName(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Returns the unique ID for a userSecurityName. This method is called when
 * creating a credential for a user.
 *
 * @parameter userSecurityName the name of the user.
 * @return the unique ID of the user. The unique ID for a user is
 *   the stringified form of some unique, registry-specific, data
 *  that serves to represent the user.  For example, for the UNIX
 *  user registry, the unique ID for a user can be the UID.
 * @exception EntryNotFoundException if userSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUniqueUserId(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Returns the name for a user given its unique ID.
 *
 * @parameter uniqueUserId the unique ID of the user.
 * @return the userSecurityName of the user.
 * @exception EntryNotFoundException if the uniqueUserID does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUserSecurityName(String uniqueUserId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Determines if the userSecurityName exists in the registry
 *
```

```
 * @parameter userSecurityName the name of the user
 * @return true if the user is valid. false otherwise
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public boolean isValidUser(String userSecurityName)
    throws CustomRegistryException,
           RemoteException;

/**
 * Gets a list of groups that match a pattern in the registry.
 * The maximum number of groups returned is defined by the limit
 * argument.
 * This method is called by the administrative console and scripting
 * (command line) to make available the groups in the registry for adding
 * them (groups) to roles.
 *
 * @parameter pattern the pattern to match. (For e.g., a* will match all
 *   groupSecurityNames starting with a)
 * @parameter limit the maximum number of groups to return.
 *  This is very useful in situations where there are thousands of
 *            groups in the registry and getting all of them at once is not
 *            practical. A value of 0 implies get all the groups and hence
 *            must be used with care.
 * @return a Result object that contains the list of groups
 *   requested and a flag to indicate if more groups exist.
 * @exception CustomRegistryException if there is any registry-specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public Result getGroups(String pattern, int limit)
    throws CustomRegistryException,
           RemoteException;

/**
 * Returns the display name for the group specified by groupSecurityName.
 *
 * This method may be called only when the group information displayed
 * (for example, the administrative console) and not used in the actual
 * authentication or authorization purposes. If there are no display names
 * in the registry return null or empty string.
 *
 * @parameter groupSecurityName the name of the group.
 * @return the display name for the group. The display name
 *   is a registry-specific string that represents a descriptive, not
 *  necessarily unique, name for a group. If a display name does
 *            not exist return null or empty string.
 * @exception EntryNotFoundException if groupSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getGroupDisplayName(String groupSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Returns the unique ID for a group.

 * @parameter groupSecurityName the name of the group.
 * @return the unique ID of the group. The unique ID for
 *  a group is the stringified form of some unique,
 *  registry-specific, data that serves to represent the group.
 *  For example, for the UNIX user registry, the unique ID might
 *  be the GID.
 * @exception EntryNotFoundException if groupSecurityName does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getUniqueGroupId(String groupSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;


/**
 * Returns the unique IDs for all the groups that contain the unique ID of
 * a user.
 * Called during creation of a user's credential.
 *
 * @parameter uniqueUserId the unique ID of the user.
 * @return a list of all the group unique IDs that the unique user ID
 *   belongs to. The unique ID for an entry is the stringified
 *  form of some unique, registry-specific, data that serves
 *  to represent the entry.  For example, for the
 *   UNIX user registry, the unique ID for a group could be the GID
 *  and the unique ID for the user could be the UID.
```

```
 * @exception EntryNotFoundException if unique user ID does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public List getUniqueGroupIds(String uniqueUserId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Returns the name for a group given its unique ID.
 *
 * @parameter uniqueGroupId the unique ID of the group.
 * @return the name of the group.
 * @exception EntryNotFoundException if the uniqueGroupId does not exist.
 * @exception CustomRegistryException if there is any registry-specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public String getGroupSecurityName(String uniqueGroupId)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Determines if the groupSecurityName exists in the registry
 *
 * @parameter groupSecurityName the name of the group
 * @return true if the groups exists, false otherwise
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public boolean isValidGroup(String groupSecurityName)
    throws CustomRegistryException,
           RemoteException;

/**
 * Returns the securityNames of all the groups that contain the user
 *
 * This method is called by administrative console and scripting
 * (command line) to verify the user entered for RunAsRole mapping belongs
 * to that role  in the roles to user mapping. Initially, the check is done
 * to see if the role contains the user. If the role does not contain the user
 * explicitly, this method is called to get the groups that this user
 * belongs to so that checks are made on the groups that the role contains.
 *
 * @parameter userSecurityName the name of the user
 * @return a List of all the group securityNames that the user
 *   belongs to.
 * @exception EntryNotFoundException if user does not exist.
 * @exception CustomRegistryException if there is any registry specific
 *            problem
 * @exception RemoteException as this extends java.rmi.Remote
 **/
 public List getGroupsForUser(String userSecurityName)
    throws EntryNotFoundException,
           CustomRegistryException,
           RemoteException;

/**
 * Gets a list of users in a group.
 *
 * The maximum number of users returned is defined by the limit
 * argument.
 *
 * This method is used by the WebSphere Business Integration
 * Server Foundation process choreographer when staff assignments
 * are modeled using groups.
 *

 * In rare situations where you are working with a user registry and it is not
 * practical to get all of the users from any of your groups (for example if
 * a large number of users exist) you can create the NotImplementedException
 * for those particular groups. Make sure that if the WebSphere Business
 * Integration Server Foundation Process Choreographer is installed (or
 * if installed later) that the users are not modeled using these particular groups.
 * If no concern exists about the staff assignments returning the users from
 * groups in the registry it is recommended that this method be implemented
 * without throwing the NotImplemented exception.
 *
 * @parameter groupSecurityName that represents the name of the group
 * @parameter limit the maximum number of users to return.
 *            This option is very useful in situations where lots of
 *            users are in the registry and getting all of them at
 *            once is not practical. A value of 0 means  get all of
 *            the users and must be used with care.
 * @return    a Result object that contains the list of users
 *            requested and a flag to indicate if more users exist.
```

```
   * @deprecated This method will be deprecated in the future.
   * @exception NotImplementedException create this exception in rare situations
   *            if it is not practical to get this information for any of the
   *            groups from the registry.
   * @exception EntryNotFoundException if the group does not exist in
   *            the registry
   * @exception CustomRegistryException if any registry-specific
   *            problem occurs
   * @exception RemoteException as this extends java.rmi.Remote interface
   **/
   public Result getUsersForGroup(String groupSecurityName, int limit)
      throws NotImplementedException,
             EntryNotFoundException,
             CustomRegistryException,
             RemoteException;

  /**
   * This method is implemented internally by the WebSphere Application Server
   * code in this release. This method is not called for the custom registry
   * implementations for this release. Return null in the implementation.
   *
   * Note that because this method is not called you can also return the
   * NotImplementedException as the previous documentation says.
   *
   **/
   public com.ibm.websphere.security.cred.WSCredential
                               createCredential(String userSecurityName)
      throws NotImplementedException,
      EntryNotFoundException,
             CustomRegistryException,
             RemoteException;
}
```

Developing a custom SAF EJB role mapper

WebSphere Application Server for z/OS allows an installation to map Java Platform, Enterprise Edition (Java EE) role names to SAF EJBRole profile names.

Before you begin

WebSphere Application Server for z/OS supports the use of a custom SAF EJB role mapper. The custom SAF EJB role mapper allows an installation to map J2EE role names to SAF EJBRole profile names. Without the SAF EJB role mapper, you must deploy an application by using a role in the deployment descriptor of a component that is identical to the name of an EJBROLE class profile. The security administrator defines EJBROLE profiles and provides the permission to these profiles to SAF users or groups.

Using SAF EJBROLE class profiles can conflict with the standard Java EE role naming conventions. Java EE role names are Unicode strings of any length. RACF® class profiles are restricted to 240 characters in length and cannot be defined if these profiles contain any white spaces or extended code page characters.

If a Java EE role name for an installation conflicts with these RACF restrictions, an installation can use the SAF EJB role mapper exit to map the desired Java EE role name to an acceptable class profile name.

The custom SAF role mapper is a Java-based exit to replace the EJBROLE class profile construction algorithm. The custom SAF role mapper is called to generate a profile for authorization and delegation requests. The role mapper passes the name of the application and the name of the role then passes back the appropriate class profile name. Information about the server name, cell name, and the SAF profile prefix (previously referred to as the z/OS security domain) is provided to the implementation during initialization.

You can set the com.ibm.websphere.security.SAF.RoleMapper custom property on the z/OS SAF authorization panel in the administrative console. You also can enable the role mapper by setting the custom property com.ibm.websphere.security.SAF.RoleMapper to the name of the class that is to be given control.

Procedure

1. Build your custom SAF role mapper. The SAFRoleMapper example (below) can be used as a reference.

```
public class SAFRoleMapperImpl1 {
 String domainPrefix = null;

 public void initialize(Properties context) {
  domainPrefix = context.get(SAFRoleMapper.DOMAIN_NAME);
 }

 public String getProfileNameFromRole(String app, String role) {
  String profile = app + "." + role;
  if (domainPrefix != null) {
   profile = domainPrefix + "." + profile;
  }
  profile = profile.replaceAll("\\%", "#");
  profile = profile.replaceAll("\\&", "#");
  profile = profile.replaceAll("\\*", "#");
  profile = profile.replaceAll("\\s","#");

  return profile;
 }
}
```

2. Click **Security > Global security > z/OS SAF authorization** and enable the role mapper by providing the name of the class that you want to give control in the **SAF profile mapper** field. You also can set this property as a custom property by entering `com.ibm.websphere.security.SAF.RoleMapper` as the name and providing the name of the class in the value field.
3. Click **Security > Global security > External authorization providers** and select the **System Authorization Facility (SAF) authorization** option to enable SAF as the authorization provider. After you select this option, click **z/OS SAF authorization** under Related items to configure the SAF authorization options.

   You also can set this property as a custom property by entering `com.ibm.websphere.security.SAF.authorization` as the name and `true` as the value.

Implementing custom password encryption

WebSphere Application Server supports the use of custom password encryption.

Before you begin

An installation can implement any password encryption algorithm it chooses.

About this task

Complete the following steps to implement custom password encryption:

Procedure

1. Build your custom password encryption class. An example of a custom password encryption class follows.

```
// CustomPasswordEncryption
// Encryption and decryption functions
public interface CustomPasswordEncryption {
    public EncryptedInfo encrypt(byte[] clearText) throws PasswordEncryptException;
    public byte[] decrypt(EncryptedInfo cipherTextInfo) throws PasswordEncryptException;
    public void initialize(HashMap initParameters);
};
// Encapsulation of cipher text and label
public class EncryptedInfo {
```

```
    public EncryptedInfo(byte[] bytes, String keyAlias);
    public byte[] getEncryptedBytes();
    public String getKeyAlias();
};
```

2. If you need to custom encode passwords in property files, manually edit the PropFilePasswordEncoder.sh or PropFilePasswordEncoder.bat file.
   a. Use a file editor to open the PropFilePasswordEncoder.sh or PropFilePasswordEncoder.bat file.
   b. Locate the following lines near the end of the file:

```
"%JAVA_HOME%/bin/java" -Dcmd.properties.file=%TMPJAVAPROPFILE%
"-Dwas.install.root=%WAS_HOME%" com.ibm.ws.bootstrap.WSLauncher
com.ibm.ws.security.util.PropFilePasswordEncoder %1 %2
```

   c. Add following lines to the call.

      These custom properties will be passed to the command so that PropFilePasswordEncoder will look for custom encoding classes and utilize it.

```
-Dcom.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled=true
-Dcom.ibm.wsspi.security.crypto.customPasswordEncryptionClass=(customEncoding class file)
```

   The updated lines should look like the following lines:

```
"%JAVA_HOME%/bin/java" -Dcmd.properties.file=%TMPJAVAPROPFILE%
-Dcom.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled=true
-Dcom.ibm.wsspi.security.crypto.customPasswordEncryptionClass=(customEncoding class file)
"-Dwas.install.root=%WAS_HOME%" com.ibm.ws.bootstrap.WSLauncher
com.ibm.ws.security.util.PropFilePasswordEncoder %1 %2
```

3. Enable custom password encryption.
   a. Set the custom property **com.ibm.wsspi.security.crypto.customPasswordEncryptionClass** to the name of the class that is to be given control.
   b. Enable the function. Set the custom property, **com.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled** to `true`.

Results

Custom password encryption at the installation is complete.

Developing applications that use programmatic security

For some applications, declarative security is not sufficient to express the security model of the application. Use this topic to develop applications that use programmatic security.

About this task

IBM WebSphere Application Server provides security components that provide or collaborate with other services to provide authentication, authorization, delegation, and data protection. WebSphere Application Server also supports the security features that are described in the Java Platform, Enterprise Edition (Java EE) specification. An application goes through three stages before it is ready to run:

- Development
- Assembly
- Deployment

Most of the security for an application is configured during the assembly stage. The security that is configured during the assembly stage is called *declarative security* because the security is *declared* or *defined* in the deployment descriptors. The declarative security is enforced by the security runtime. For some applications, declarative security is not sufficient to express the security model of the application. For these applications, you can use *programmatic security*.

Procedure

1. Develop secure web applications. For more information, see “Developing with programmatic security APIs for web applications” on page 909.

2. Develop servlet filters for form login processing. For more information, see “Developing servlet filters for form login processing” on page 924.
3. Develop form login pages. For more information, see “Customizing web application login” on page 921.
4. Develop enterprise bean component applications. For more information, see “Developing with programmatic APIs for EJB applications” on page 917.
5. Develop with Java Authentication and Authorization Service to log in programmatically.

   For more information, see topics about developing programmatic logins with the Java Authentication and Authorization Service.
6. Develop your own Java EE security mapping module.

   For more information, see topics about configuring programmatic logins for Java Authentication and Authorization Service.
7. Develop custom user registries. For more information, see “Developing stand-alone custom registries” on page 877.
8. Develop a custom interceptor for trust associations.

Protecting system resources and APIs (Java 2 security) for developing applications

Java 2 security is a programming model that is very pervasive and has a huge impact on application development.

Before you begin

Java 2 security is orthogonal to Java Platform, Enterprise Edition (Java EE) role-based security; you can disable or enable it independently of administrative security.

However, it does provide an extra level of access control protection on top of the Java EE role-based authorization. It particularly addresses the protection of system resources and application programming interfaces (API). Administrators need to consider the benefits against the risks of disabling Java 2 security.

The following recommendations are provided to help enable Java 2 security in a test or production environment:

1. Make sure the application is developed with the Java 2 security programming model. Developers have to know whether or not the APIs that are used in the applications are protected by Java 2 security. It is very important that the required permissions for the APIs used are declared in the policy file (`was.policy`), or the application fails to run when Java 2 security is enabled. Developers can reference the website for Development Kit APIs that are protected by Java 2 security. See the Programming model and decisions section of the Security: Resources for Learning topic to visit this website.
2. Make sure that migrated applications from previous releases are given the required permissions. Because Java 2 security is not supported or partially supported in previous WebSphere Application Server releases, applications developed prior to Version 5 most likely are not using the Java 2 security programming model. No easy way to find out all the required permissions for the application is available. The following are activities you can perform to determine the extra permissions that are required by an application:
   - Code review and code inspection
   - Application documentation review
   - Sandbox testing of migrated enterprise applications with Java 2 security enabled in a preproduction environment. Enable tracing in WebSphere Java 2 security manager to help determine the missing permissions in the application policy file. The trace specification is: `com.ibm.ws.security.core.SecurityManager=all=enabled`.
   - Use the com.ibm.websphere.java2secman.norethrow system property to aid debugging. Do not use this property in a production environment.

The default permission set for applications is the recommended permission set that is defined in the J2EE 1.3 Specification. The default is declared in the *app_server_root*/`profiles/`*profile_name*`/config/cells/`

cell_name/nodes/*node_name*/app.policy policy file with permissions defined in the Development Kit (*JAVA_HOME*/jre/lib/security/java.policy) policy file that grant permissions to everyone. However, applications are denied permissions that are declared in the profiles/*profile_name*/config/cells/*cell_name*/filter.policy file. Permissions that are declared in the *filter.policy* file are filtered for applications during the permission check.

Define the required permissions for an application in a was.policy file and embed the was.policy file in the application enterprise archive (EAR) file as *YOURAPP.ear/META-INF/was.policy*, see "Configuring Java 2 security policy files" on page 890 for details.

The following steps describe how to enforce Java 2 security on the cell level for WebSphere Application Server, Network Deployment and the server level for WebSphere Application Server, Express

Procedure

1. Click **Security > Global security**. The Global security panel is displayed.
2. Select the **Use Java 2 security to restrict application access to local resources** option.
3. Click **OK** or **Apply**.
4. Click **Save** to save the changes.
5. Restart the server for the changes to take effect.

Results

Java 2 security is enabled and enforced for the servers. Java 2 security permission is selected when a Java 2 security protected API is called.

When to use Java 2 security

1. Enable protection on system resources, for example when opening or listening to a socket connection, reading or writing to operating system file systems, reading or writing Java virtual machine system properties, and so on.
2. Prevent application code from calling destructive APIs, for example, calling the System.exit method brings down the application server.
3. Prevent application code from obtaining privileged information (passwords) or gaining extra privileges (obtaining server credentials).

What to do next

You can enforce Java 2 security on the server level for WebSphere Application Server, Network Deployment by completing the following steps.

Note: Changes to Java 2 security settings on the server level override the settings on the cell level.

1. Click **Servers > Application servers >** *server_name*.
2. Under Security, click **Server security**.
3. Select the **Security settings for this server override cell settings** option.
4. Select the **Use Java 2 security to restrict application access to local resources** option.
5. Click **OK** or **Apply**.
6. Click **Save** to save the changes.
7. Restart the server for the changes to take effect.

The Java 2 security manager is enhanced to dump the Java 2 security permissions that are granted to all classes on the call stack when an application is denied access to a resource. The java.security.AccessControlException exception is created. However, this tracing capability is disabled by default. You can enable this capability by specifying the server trace service with the com.ibm.ws.security.core.SecurityManager=all=enabled trace specification. When the exception is

created, the trace dump provides hints to determine whether the application is missing permissions or the product runtime code or the third-party libraries that are used are not properly marked as `privileged` when accessing Java 2 security-protected resources.

Using PolicyTool to edit policy files for Java 2 security:

Use the **PolicyTool** utility to update policy files.

Before you begin

Java 2 security uses several policy files to determine the granted permission for each Java program. The Java Development Kit provides the **PolicyTool** tool to edit these policy files. This tool is recommended for editing any policy file to verify the syntax of its contents. Syntax errors in the policy file cause an AccessControlException exception when the application runs, including the server start. Identifying the cause of this exception is not easy because the user might not be familiar with the resource that has an access violation. Be careful when you edit these policy files.

To use the **PolicyTool** utility with WebSphere Application Server for z/OS, choose one of the following two options:

- Copy the policy files to another platform such as Microsoft Windows and modify the files. To use this option, you must issue the FTP command to transfer the files to the other platform, invoke the **PolicyTool**, and transfer the updated files back to the z/OS system in binary mode.
- Invoke the **PolicyTool** that is supplied with the Software Development Kit (SDK) installed on your z/OS system.

Procedure

1. Invoke the **PolicyTool** that is supplied with the Software Development Kit (SDK) installed on your z/OS system.
   a. Export the display to an Xwindows-enabled device. For example, in Open MVS™ (OMVS), type `export DISPLAY=<IP_address_of_the_Xwindows_device>:0.0`
   b. Enable the z/OS system to access the display of the Xwindows-enabled device. For example, on AIX® systems, type `xhost + address_of_the_MVS_system`.
   c. Convert the policy file to the Extended Binary Coded Decimal Interchange Code (EBCDIC) format.
   d. Invoke the **PolicyTool** on OMVS by typing `$JAVA_HOME/policytool`. The *JAVA_HOME* variable represents the directory in which the SDK is installed.
2. Click **File** > **Open**.
3. Navigate the directory tree in the **Open** window to pick up the policy file that you need to update. After selecting the policy file, click **Open**. The code base entries are listed in the window.
4. Create or modify the code base entry.
   a. Modify the existing code base entry by double-clicking the code base, or click the code base and click **Edit Policy Entry**. The Policy Entry window opens with the permission list defined for the selected code base.
   b. Create a new code base entry by clicking **Add Policy Entry**.

      The Policy Entry window opens. At the code base column, enter the code base information as a URL format.

      For example, you can enter:

```
app_server_root/InstalledApps/testcase.ear
```

      where the *app_server_root* variable represents your installation location.
5. Modify or add the permission specification.

a. Modify the permission specification by double-clicking the entry that you want to modify, or by selecting the permission and clicking **Edit Permission**. The Permissions window opens with the selected permission information.
b. Add a new permission by clicking **Add Permission**. The Permissions window opens. In the Permissions window are four rows for Permission, Target Name, Actions, and Signed By.

6. Select the permission from the Permission list. The selected permission displays. After a permission is selected, the Target Name, Actions, and Signed By fields automatically show the valid choices or they enable text input in the right text input area.
   a. Select **Target Name** from the list, or enter the target name in the right text input area.
   b. Select **Actions** from the list.
   c. Input **Signed By** if it is needed.

      Important: The Signed By keyword is not supported in the following policy files: `app.policy`, `spi.policy`, `library.policy`, `was.policy`, and `filter.policy` files. However, the Signed By keyword is supported in the following policy files: `#java.policy`, `server.policy`, and `client.policy` files. The Java Authentication and Authorization Service (JAAS) is not supported in the `app.policy`, `spi.policy`, `library.policy`, `was.policy`, and `filter.policy` files. However, the JAAS principal keyword is supported in a JAAS policy file when it is specified by the java.security.auth.policy Java virtual machine (JVM) system property.
7. Click **OK** to close the Permissions window. Modified permission entries of the specified code base display.
8. Click **Done** to close the window. Modified code base entries are listed. Repeat the previous steps until you complete editing.
9. Click **File** > **Save** after you finish editing the file.
10. Convert the policy file back from the EBCDIC format to the ASCII format.

Results

A policy file is updated. If any policy files need editing, use the **PolicyTool** utility. Do not edit the policy file manually. Syntax errors in the policy files can potentially cause application servers or enterprise applications to not start or function incorrectly. For the changes in the updated policy file to take effect, restart the Java processes.

Configuring Java 2 security policy files:

Users can configure Java 2 security policy files so that the required permission is granted for the specified WebSphere Application Server enterprise application.

Before you begin

Java 2 security uses several policy files to determine the permissions for each Java programs.

See the Java 2 security policy files topic for the list of available policy files that are supported by WebSphere Application Server.

Two types of policy files are supported by WebSphere Application Server: dynamic policy files and static policy files. Static policy files provide the default permissions. Dynamic policy files provide application permissions. Six dynamic policy files are provided:

Table 93. Dynamic policy files. This table lists the dynamic policy files.

Policy file name	Description
app.policy	Contains default permissions for all of the enterprise applications in the cell. **Note:** Updates to the app.policy file only apply to the enterprise applications on the node to which the app.policy file belongs.
was.policy	Contains application-specific permissions for an WebSphere Application Server enterprise application. This file is packaged in an enterprise archive (EAR) file.
ra.xml	Contains connector application specific permissions for a WebSphere Application Server enterprise application. This file is packaged in a resource adapter archive (RAR) file.
spi.policy	Contains permissions for Service Provider Interface (SPI) or third-party resources that are embedded in WebSphere Application Server. The default contents grant everything. Update this file carefully when the cell requires more protection against SPI in the cell. This file is applied to all of the SPIs that are defined in the resources.xml file.
library.policy	Contains permissions for the shared library of enterprise applications.
filter.policy	Contains the list of permissions that require filtering from the was.policy file and the app.policy file in the cell. This filtering mechanism only applies to the was.policy and app.policy files.

In WebSphere Application Server, applications must have the appropriate thread permissions specified in the was.policy or app.policy file. Without the thread permissions specified, the application cannot manipulate threads and WebSphere Application Server creates a java.security.AccessControlException exception. The app.policy file applies to a specified node. If you change the permissions in one app.policy file, you must incorporate the new thread policy in the same file on the remaining nodes. Also, if you add the thread permissions to the app.policy file, you must restart WebSphere Application Server to enforce the new permissions. However, if you add the permissions to the was.policy file for a specific application, you do not need to restart WebSphere Application Server. An administrator must add the following code to a was.policy or app.policy file for an application to manipulate threads:

```
grant codeBase "file:${application}" {
  permission java.lang.RuntimePermission "stopThread";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
};
```

Important: The Signed By keyword is not supported in the following policy files: app.policy, spi.policy, library.policy, was.policy, and filter.policy files. However, the Signed By keyword is supported in the following policy files:java.policy, server.policy, and client.policy files. The Java Authentication and Authorization Service (JAAS) is not supported in the app.policy, spi.policy, library.policy, was.policy, and filter.policy files. However, the JAAS principal keyword is supported in a JAAS policy file when it is specified by the java.security.auth.policy Java virtual machine (JVM) system property. You can statically set the authorization policy files in java.security.auth.policy with auth.policy.url.n=*URL*, where *URL* is the location of the authorization policy.

Procedure

1. Identify the policy file to update.
   - If the permission is required by an application, update the static policy file. Refer to “Configuring static policy files in Java 2 security” on page 903.
   - If the permission is required by all of the WebSphere Application Server enterprise applications in the node, refer to “spi.policy file permissions” on page 899.
   - If the permission is required only by specific WebSphere Application Server enterprise applications and the permission is required only by connector, update the ra.xml file. Refer to Refer to the Assembling resource adapter (connector) modules article for more information. Otherwise, update the was.policy file. Refer to “Configuring the was.policy file for Java 2 security” on page 896 and “Adding the was.policy file to applications for Java 2 security” on page 901.

- If the permission is required by shared libraries, refer to “library.policy file permissions” on page 900.
- If the permission is required by SPI libraries, refer to “spi.policy file permissions” on page 899.

Tip: Pick up the policy file with the smallest scope. You can avoid giving an extra permission to the Java programs and protect the resources. You can update the `ra.xml` file or the `was.policy` file rather than the `app.policy` file. Use specific component symbols ($(ejbcomponent), ${webComponent},${connectorComponent} and ${jars}) than ${application} symbols. Update dynamic policy files, rather than static policy files.

Add any permission that you never want granted to the WebSphere Application Server enterprise application in the cell to the `filter.policy` file. Refer to “filter.policy file permissions” on page 894.

2. Restart the WebSphere Application Server enterprise application.

Results

The required permission is granted for the specified WebSphere Application Server enterprise application.

app.policy file permissions:

Java 2 security uses several policy files to determine the granted permissions for each Java program. The union of the permissions that are contained in these following files is applied to the WebSphere Application Server enterprise application. This union determines the granted permissions.

For the list of available policy files that are supported by WebSphere Application Server, see the topic about Java 2 security policy files. The `app.policy` file is a default policy file that is shared by all of the WebSphere Application Server enterprise applications. The union of the permissions that are contained in the following files is applied to the WebSphere Application Server enterprise application:

- Any policy file that is specified in the policy.url.* properties in the `java.security` file.
- The `app.policy` files, which are managed by configuration and file replication services.
- The `server.policy` file.
- The `java.policy` file.
- The application `was.policy` file.
- The permission specification of the `ra.xml` file.
- The shared library, which is the `library.policy` file.

Changes made in these files are replicated to other nodes in the WebSphere Application Server, Network Deployment cell.

In WebSphere Application Server, applications that manipulate threads must have the appropriate thread permissions specified in the `was.policy` or `app.policy` file. Without the thread permissions specified, the application cannot manipulate threads and WebSphere Application Server creates a java.security.AccessControlException exception. If an administrator adds thread permissions to the `app.policy` file, the permission change requires a restart of the WebSphere Application Server. An administrator must add the following code to a `was.policy` or `app.policy` file for an application to manipulate threads:

```
grant codeBase "file:${application}" {
  permission java.lang.RuntimePermission "stopThread";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
};
```

Important: The Signed By and the Java Authentication and Authorization Service (JAAS) principal keywords are not supported in the `app.policy` file. However, the Signed By keyword is supported in the following files: `java.policy`, `server.policy`, and the `client.policy` files. The JAAS principal keyword is supported in a JAAS policy file when it is specified by the `java.security.auth.policy` Java virtual machine (JVM) system property. You can statically

set the authorization policy files in the java.security.auth.policy property with auth.policy.url.n=URL where URL is the location of the authorization policy.

If the default permissions for enterprise applications (the union of the permissions that is defined in the java.policy file, the server.policy file and the app.policy file) are enough; no action is required. The default app.policy file is used automatically. If a specific change is required to all of the enterprise applications in the cell, update the app.policy file. Syntax errors in the policy files cause start failures in the application servers. Edit these policy files carefully.

Note: Updates to the app.policy file only apply to the enterprise applications on the node to which the app.policy file belongs.

To extract the policy file, use a command prompt to enter the following command on one line using the appropriate variable values for your environment:

```
wsadmin> set obj [$AdminConfig extract  cells/cell_name/node/node_name/app.policy /temp/test/app.policy]
```

Edit the extracted app.policy file with the Policy Tool. For more information, see "Using PolicyTool to edit policy files for Java 2 security" on page 889. Changes to the app.policy file are local for the node.

To check in the policy file, use a command prompt to enter the following command on one line using the appropriate variable values for your environment:

```
wsadmin> $AdminConfig checkin cells/cell_name/nodes/node_name/app.policy  temp/test/app.policy $obj
```

Table 94. Symbols used to associate permission lists to a specific type of resource. Several product-reserved symbols are defined to associate the permission lists to a specific type of resource.

Symbol	Meaning
file:${application}	Permissions apply to all resources within the application
file:${jars}	Permissions apply to all utility Java archive (JAR) files within the application
file:${ejbComponent}	Permissions apply to enterprise bean resources within the application
file:${webComponent}	Permissions apply to web resources within the application
file:${connectorComponent}	Permissions apply to connector resources both within the application and within stand-alone connector resources.

Table 95. Symbols provided to specify the path and name for the java.io.FilePermission permission. Five embedded symbols are provided to specify the path and name for the java.io.FilePermission permission. These symbols enable flexible permission specifications. The absolute file path is fixed after the installation of the application.

Symbol	Meaning
${app.installed.path}	Path where the application is installed
${was.module.path}	Path where the module is installed
${current.cell.name}	Current cell name
${current.node.name}	Current node name
${current.server.name}	Current server name

Tip: You cannot use the ${was.module.path} in the ${application} entry.

The app.policy file supplied by WebSphere Application Server is located in the *profile_root*/config/cells/cell_name/nodes/node_name/app.policy, which contains the following default permissions:

Attention: In the following code sample, the first two lines that are related to java.io.FilePermission permission are split into two lines for illustrative purposes only.

```
grant codeBase "file:${application}" {
  // The following are required by JavaMail
  permission java.io.FilePermission "${was.install.root}${/}lib${/}activation-impl.jar", "read";
  permission java.io.FilePermission "${was.install.root}${/}lib${/}mail-impl.jar", "read";
};

grant codeBase "file:${jars}" {
```

```
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

grant codeBase "file:${connectorComponent}" {
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

grant codeBase "file:${webComponent}" {
  permission java.io.FilePermission "${was.module.path}${/}-", "read, write";
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

grant codeBase "file:${ejbComponent}" {
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};
```

If all of the WebSphere Application Server enterprise applications in a cell require permissions that are not defined as defaults in the `java.policy` file, the `server.policy` file and the `app.policy` file, then update the `app.policy` file. The symptom of a missing permission is the `java.security.AccessControlException` exception.

Note: Updates to the app.policy file only apply to the enterprise applications on the node to which the app.policy file belongs.

When a Java program receives this exception and adding this permission is justified, add a permission to the `server.policy` file, for example:

The previous permission information lines are split for the illustration. You actually enter the permission on one line.

To decide whether to add a permission, refer to the AccessControlException topic.

Restart all WebSphere Application Server enterprise applications to ensure that the updated `app.policy` file takes effect.

filter.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program. Java 2 security policy filtering is only in effect when Java 2 security is enabled.

Before modifying the `filter.policy` file, you must start the wsadmin tool.

Refer to “Protecting system resources and APIs (Java 2 security) for developing applications” on page 887. The filtering policy defined in the `filter.policy` file is cell wide. The `filter.policy` file is the only policy file that is used when restricting the permission instead of granting permission. The permissions that are listed in the filter policy file are filtered out from the `app.policy` file and the `was.policy` file. Permissions that are defined in the other policy files are not affected by the `filter.policy` file.

When a permission is filtered out, an audit message is logged. However, if the permissions that are defined in the `app.policy` file and the `was.policy` file are compound permissions like the java.security.AllPermission permission, for example, the permission is not removed. A warning message is logged. If the Issue Permission Warning flag is enabled (default) and if the `app.policy` file and the `was.policy` file contain custom permissions (non-Java API permission, the permission package name begins with characters other than `java` or `javax`), a warning message is logged and the permission is not removed. You can change the value of the `Warn if applications are granted custom permissions` option on the Global security panel. It is not recommended that you use the `AllPermission` permission for the enterprise application.

Some default permissions that are defined in the `filter.policy` file. These permissions are the minimal ones that are recommended by the product. If more permissions are added to the `filter.policy` file, certain operations can fail for enterprise applications. Add permissions to the `filter.policy` file carefully.

You cannot use the Policy Tool to edit the `filter.policy` file. Editing must be completed in a text editor. Be careful and verify that no syntax errors exist in the `filter.policy` file. If any syntax errors exist in the `filter.policy` file, the file is not loaded by the product security runtime, which implies that filtering is disabled.

To extract the `filter.policy` file, enter the following command using information from your environment:

```
set obj [$AdminConfig extract cells/cell_name/filter.policy /temp/test/filter.policy]
```

To check in the policy file, enter the following command using information from your environment:

```
$AdminConfig checkin cells/cell_name/filter.policy /temp/test/filter.policy $obj
```

An updated `filter.policy` file is applied to all of the WebSphere Application Server enterprise applications after the servers are restarted. The `filter.policy` file is managed by configuration and file replication services.

Changes made in the file are replicated to other nodes in the cell.

The `filter.policy` file that is supplied by WebSphere Application Server resides at: *app_server_root*/`profiles/profile_name/config/cells/cell_name/filter.policy`.

This fill contains these permissions as defaults:

```
filterMask {
permission java.lang.RuntimePermission "exitVM";
permission java.lang.RuntimePermission "setSecurityManager";
permission java.security.SecurityPermission "setPolicy";
permission javax.security.auth.AuthPermission "setLoginConfiguration"; };
runtimeFilterMask {
permission java.lang.RuntimePermission "exitVM";
permission java.lang.RuntimePermission "setSecurityManager";
permission java.security.SecurityPermission "setPolicy";
permission javax.security.auth.AuthPermission "setLoginConfiguration"; };
```

The permissions that are defined in `filterMask` filter are for static policy filtering. The security runtime tries to remove the permissions from applications during application startup. Compound permissions are not removed, but are issued with a warning, and application deployment is stopped if applications contain permissions that are defined in the `filterMask` filter, and if scripting is used. The runtimeFilterMask filter defines permissions that are used by the security runtime to deny access to those permissions to application thread. Do not add more permissions to the runtimeFilterMask filter. Application start failure or incorrect functioning might result. Be careful when adding more permissions to the runtimeFilterMask filter. Usually, you only need to add permissions to the `filterMask` stanza.

WebSphere Application Server relies on the filter policy file to restrict or disallow certain permissions that can compromise the integrity of the system. For instance, WebSphere Application Server considers the exitVM and setSecurityManager permissions as those permissions that most applications never have. If these permissions are granted, the following scenarios are possible:

exitVM
A servlet, JavaServer Pages (JSP) file, enterprise bean, or other library that is used by the aforementioned might call the System.exit API and cause the entire WebSphere Application Server process to terminate.

setSecurityManager
An application might install its own security manager and either grant more permissions or bypass the default policy that the WebSphere Application Server security manager enforces.

Important: In application code, do not use the setSecurityManager permission to set a security manager. When an application uses the setSecurityManager permission, a conflict exists with the internal security manager within WebSphere Application Server. If you must set a security manager in an application for Remote Method Invocation (RMI) purposes, you also must select the Use Java 2 security to restrict application access to local resources option on the Global security panel within the WebSphere Application Server administrative console. WebSphere Application Server then registers a security manager, which the application code can verify is registered by using the System.getSecurityManager application programming interface (API).

For the updated `filter.policy` file to take effect, restart related Java processes.

Configuring the was.policy file for Java 2 security:

You should update the `was.policy` file if the application has specific resources to access.

Before you begin

Java 2 security uses several policy files to determine the granted permission for each Java program. The `was.policy` file is an application-specific policy file for WebSphere Application Server enterprise applications. This file is embedded in the `META-INF/was.policy` enterprise archive (.EAR) file. The `was.policy` file is located in:

profile_root`/config/cells/`*cell_name*`/applications/`
ear_file_name`/deployments/`*application_name*`/META-INF/was.policy`

See Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server Version 6.1.

The union of the permissions that are contained in the following files is applied to the WebSphere Application Server enterprise application:

- Any policy file that is specified in the policy.url.* properties in the `java.security` file.
- The `app.policy` files, which are managed by configuration and file replication services.
- The `server.policy` file.
- The `java.policy` file.
- The application `was.policy` file.
- The permission specification of the `ra.xml` file.
- The shared library, which is the `library.policy` file.

Changes made in these files are replicated to other nodes in the cell.

Table 96. Symbols defined to associate permission lists to a specific type of resource. Several product-reserved symbols are defined to associate the permission lists to a specific type of resource.

Symbol	Definition
file:${application}	Permissions apply to all resources used within the application.
file:${jars}	Permissions apply to all utility Java archive (JAR) files within the application

Table 96. Symbols defined to associate permission lists to a specific type of resource (continued). Several product-reserved symbols are defined to associate the permission lists to a specific type of resource.

Symbol	Definition
file:${ejbComponent}	Permissions apply to enterprise bean resources within the application
file:${webComponent}	Permissions apply to web resources within the application
file:${connectorComponent}	Permissions apply to connector resources within the application

In WebSphere Application Server, applications that manipulate threads must have the appropriate thread permissions specified in the `was.policy` or `app.policy` file. Without the thread permissions specified, the application cannot manipulate threads and WebSphere Application Server creates a java.security.AccessControlException exception. If you add the permissions to the `was.policy` file for a specific application, you do not need to restart WebSphere Application Server. An administrator must add the following code to a `was.policy` or `app.policy` file for an application to manipulate threads:

```
grant codeBase "file:${application}" {
  permission java.lang.RuntimePermission "stopThread";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "modifyThreadGroup";
};
```

An administrator can add the thread permissions to the `app.policy` file, but the permission change requires a restart of WebSphere Application Server.

Important: The Signed By and the Java Authentication and Authorization Service (JAAS) principal keywords are not supported in the `was.policy` file. The Signed By keyword is supported in the `java.policy`, `server.policy`, and `client.policy` policy file. The JAAS principal keyword is supported in a JAAS policy file when it is specified by the `java.security.auth.policy` Java virtual machine (JVM) system property. You can statically set the authorization policy files in the `java.security.auth.policy` file with the `auth.policy.url.n=`*URL*, where *URL* is the location of the authorization policy.

Other than these blocks, you can specify the module name for granular settings. For example,

```
grant codeBase "file:DefaultWebApplication.war" {
  permission java.security.SecurityPermission "printIdentity";
};

grant codeBase "file:IncCMP11.jar" {
  permission java.io.FilePermission
      "${user.install.root}${/}bin${/}DefaultDB${/}-",
      "read,write,delete";
};
```

Table 97. Embedded symbols provided to specify the path and name for the java.io.FilePermission permission. Five embedded symbols are provided to specify the path and name for the java.io.FilePermission permission. These symbols enable flexible permission specification. The absolute file path is fixed after the application is installed.

Symbol	Definition
${app.installed.path}	Path where the application is installed
${was.module.path}	Path where the module is installed
${current.cell.name}	Current® cell name
${current.node.name}	Current node name
${current.server.name}	Current server name

About this task

If the default permissions for the enterprise application are enough, an action is not required. The default permissions are a union of the permissions that are defined in the `java.policy` file, the `server.policy` file, and the `app.policy` file. If an application has specific resources to access, update the `was.policy` file. The first two steps assume that you are creating a new policy file.

Tip: Syntax errors in the policy files cause the application server to fail. Use care when editing these policy files.

Procedure

1. Create or edit a new `was.policy` file by using the PolicyTool. For more information, see “Using PolicyTool to edit policy files for Java 2 security” on page 889.
2. Package the `was.policy` file into the enterprise archive (EAR) file.

   For more information, see “Adding the was.policy file to applications for Java 2 security” on page 901.The following instructions describe how to import a `was.policy` file.

   a. Import the EAR file into an assembly tool.

   b. Open the Project Navigator view.

   c. Expand the EAR file and click **META-INF**. You might find a `was.policy` file in the META-INF directory. If you want to delete the file, right-click the file name and select **Delete**.

   d. At the bottom of the Project Navigator view, click **J2EE Hierarchy**.

   e. Import the `was.policy` file by right-clicking the **Modules** directory within the deployment descriptor and by clicking **Import > Import > File system**.

   f. Click **Next**.

   g. Enter the path name to the `was.policy` file in the **From directory** field or click **Browse** to locate the file.

   h. Verify that the path directory that is listed in the **Into directory** field lists the correct `META-INF` directory.

   i. Click **Finish**.

   j. To validate the EAR file, right-click the EAR file, which contains the Modules directory, and click **Run Validation**.

   k. To save the new EAR file, right-click the EAR file, and click **Export > Export EAR file**. If you do not save the revised EAR file, the EAR file will contain the new `was.policy` file. However, if the workspace becomes corrupted, you might lose the revised EAR file.

   l. To generate deployment code, right-click the EAR file and click **Generate Deployment Code**.
3. Update an existing installed application, if one already exists. Modify the `was.policy` file with the Policy Tool. For more information, see “Using PolicyTool to edit policy files for Java 2 security” on page 889.

   a. Extract the policy file. Enter the following command from a command prompt:

```
wsadmin> set obj [$AdminConfig extract profiles/profile_name/cells/cell_name
/application/ear_file_name/deployments/application_name
/META_INF/was.policy c:/temp/test/was.policy]
```

   Enter the three previous lines as one continuous line. They are display here for illustration only.

   b. Edit the extracted `was.policy` file with the Policy Tool. For more information, see “Using PolicyTool to edit policy files for Java 2 security” on page 889.

   c. Check in the policy file. Enter the following at a command prompt:

```
wsadmin> $AdminConfig checkin profiles/profile_name/cells/cell_name/application/
ear_file_name/deployments/application_name/META_INF/was.policy
c:/temp/test/was.policy $obj
```

   Enter the three previous lines as one continuous line. They are display here for illustration only.

Results

The updated `was.policy` file is applied to the application after the application restarts.

Example

```
java.security.AccessControlException: access denied (java.io.FilePermission
${was.install.root}/java/ext/mail.jar read)
```

If an application must access a specific resource that is not defined as a default in the java.policy file, the server.policy file, and the app.policy, delete the was.policy file for that application. The symptom of the missing permission is the java.security.AccessControlException exception. The missing permission is listed in the exception data:

Note: Examples that appear below are split into several lines for illustration only. You actually enter the permission on one line.

```
java.security.AccessControlException: access denied (java.io.FilePermission
${was.install.root}/java/ext/mail.jar read)
```

When a Java program receives this exception and adding this permission is justified, add the following permission to the was.policy file:

```
grant codeBase "file:user_client_installed_location" {
  permission java.io.FilePermission
"${was.install.root}$(/)java$(/)jre$(/)lib$(/)ext$(/)mail.jar", "read";
};
```

To determine whether to add a permission, see Access control exception for Java 2 security.

What to do next

Restart all applications for the updated app.policy file to take effect.

spi.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

For the list of available policy files that are supported by WebSphere Application Server Version 6.0.x, see Java 2 security policy files.

Because the default permission for the Service Provider Interface (SPI) is the AllPermission permission, the only reason to update the spi.policy file is a restricted SPI permission. When a change in the spi.policy is required, complete the following steps.

Syntax errors in the policy files cause the application server to fail. Edit these policy files carefully.

Important: Do not place the codebase keyword or any other keyword after the filterMask and runtimeFilterMask keywords. The Signed By and the Java Authentication and Authorization Service (JAAS) Principal keywords are not supported in the spi.policy file. The Signed By keyword is supported in the java.policy, server.policy, and client.policy policy files. The JAAS Principal keyword is supported in a JAAS policy file that is specified by the java.security.auth.policy Java virtual machine (JVM) system property. You can statically set the authorization policy files in java.security.auth.policy with auth.policy.url.n=URL, where URL is the location of the authorization policy.

To extract the filter.policy file, enter the following command using information from your environment:

set obj [$AdminConfig extract profiles/profile_name/cells/cell_name/nodes/node_name/spi.policy c:/temp/test/spi.policy]

Edit the file using the Policy Tool. For more information, see “Using PolicyTool to edit policy files for Java 2 security” on page 889.

To check in the policy file, enter the following command using information from your environment:

The updated spi.policy is applied to the Service Provider Interface (SPI) libraries after the Java process is restarted.

$AdminConfig checkin profiles/profile_name/cells/cell_name/nodes/node_name/spi.policy c:/temp/test/spi.policy $obj

Examples

The spi.policy file is the template for SPIs or third-party resources embedded in the product. Examples of SPIs are Java Message Services (JMS) (MQSeries) and Java database connectivity (JDBC) drivers. They are specified in the resources.xml file. The dynamic policy grants the permissions that are defined in the spi.policy file to the class paths defined in the resources.xml file. The union of the permission that is contained in the java.policy file and the spi.policy file are applied to the SPI libraries. The spi.policy files are managed by configuration and file replication services.

Changes made in these files are replicated to other nodes in the cell.

You can find the spi.policy file that is supplied by WebSphere Application Server in the following location: *app_server_root*/profiles/profile_name/config/cells/cell_name/nodes/node_name/spi.policy. This file contains the following default permission:

```
grant {
  permission java.security.AllPermission;
};
```

Restart the related Java processes for the changes in the spi.policy file to become effective.

library.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

For the list of available policy files that are supported by WebSphere Application Server, see Java 2 security policy files.

The library.policy file is the template for shared libraries (Java library classes). Multiple enterprise applications can define and use shared libraries. Refer to Managing shared libraries for information on how to define and manage the shared libraries.

If the default permissions for a shared library (union of the permissions defined in the java.policy file, the app.policy file and the library.policy file) are enough, no action is required. The default library policy is picked up automatically. If a specific change is required to share a library in the cell, update the library.policy file.

Syntax errors in the policy files cause the application server to fail. Edit these policy files carefully.

Important: Do not place the codebase keyword or any other keyword after the grant keyword. The Signed By keyword and the Java Authentication and Authorization Service (JAAS) Principal keyword are not supported in the library.policy file. The Signed By keyword is supported in the java.policy, the server.policy, and the client.policy policy files. The JAAS Principal keyword is supported in a JAAS policy file when it is specified by the Java virtual machine (JVM) system property, java.security.auth.policy. You can statically set the authorization policy files in the java.security.auth.policy file with auth.policy.url.n=URL where URL is the location of the authorization policy.

To extract the policy file, use a command prompt to enter the following command using the appropriate variable values for your environment:The previous two lines were split onto two lines for illustrative purposes only.

wsadmin> set obj [$AdminConfig extract cells/cell_name/nodes/
node_name/library.policy /temp/test/library.policy]

Edit the extracted `library.policy` file with the Policy Tool. For more information, see “Using PolicyTool to edit policy files for Java 2 security” on page 889.

To check in the policy file, use a command prompt to enter the following command using the appropriate variable values for your environment:An updated `library.policy` is applied to shared libraries after the servers restart.

wsadmin> $AdminConfig checkin cells/cell_name/nodes/node_name/library.policy
temp/test/library.policy $obj

Example

The union of the permission that is contained in the `java.policy` file, the `app.policy` file, and the `library.policy` file are applied to the shared libraries. The `library.policy` file is managed by configuration and file replication services.

Changes made in the file are replicated to other nodes in the cell.

The `library.policy` file are supplied by WebSphere Application Server resides at: *app_server_root*/`config/cells/cell_name/nodes/node_name/` directory. The file contains an empty permission entry as a default. For example:

```
grant {
  };
```

If the shared library in a cell requires permissions that are not defined as defaults in the`java.policy` file, the `app.policy` file and the `library.policy` file, update the `library.policy` file. The missing permission causes the java.security.AccessControlException exception. The missing permission is listed in the exception data.

When a Java program receives this exception and adding this permission is justified, add a permission to the `library.policy` file.

To decide whether to add a permission, refer to Access control exception for Java 2 security.

Restart the related Java processes for the changes in the `library.policy` file to become effective.

Adding the was.policy file to applications for Java 2 security:

An application might need a `was.policy` file if it accesses resources that require more permissions than those granted in the default `app.policy` file.

About this task

When Java 2 security is enabled for a WebSphere Application Server, all the applications that run on WebSphere Application Server undergo a security check before accessing system resources. An application might need a `was.policy` file if it accesses resources that require more permissions than those granted in the default `app.policy` file. By default, the product security reads an `app.policy` file that is located in each node and grants the permissions in the `app.policy` file to all the applications. Include any additional required permissions in the `was.policy` file. The `was.policy` file is only required if an application requires additional permissions.

The default policy file for all applications is specified in the app.policy file. This file is provided by the product security, is common to all applications, and you do not change this file. Add any new permissions that are required for an application in the was.policy file.

The app.policy file supplied by WebSphere Application Server resides at *app_server_root*/config/cells/profile/profile_name/config/*cell_name*/nodes/*node_name*/app.policy. The contents of the app.policy file are presented in the following example:

Attention: In the following code sample, the two permissions that are required by JavaMail are split onto two lines for illustration only. You actually enter the permission on one line.

```
// The following permissions apply to all the components under the application.

grant codeBase "file:${application}" {
   // The following are required by JavaMail

  permission java.io.FilePermission "
        ${was.install.root}${/}lib${/}activation-impl.jar",
"read";

  permission java.io.FilePermission "
        ${was.install.root}${/}lib${/}mail-impl.jar","read";

  };
   // The following permissions apply to all utility .jar files  (other
   //  than enterprise beans JAR files) in the application.
grant codeBase "file:${jars}" {
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

// The following permissions apply to connector resources within the application
grant codeBase "file:${connectorComponent}" {
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

// The following permissions apply to all the web modules (.war files)
// within the application.
grant codeBase "file:${webComponent}" {
  permission java.io.FilePermission "${was.module.path}${/}-", "read, write";
       //  where "was.module.path" is the path where the web module is
       //  installed. Refer to Dynamic policy concepts for other symbols.
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.util.PropertyPermission "*", "read";
};

// The following permissions apply to all the EJB modules within the application.
grant codeBase "file:${ejbComponent}" {
 permission java.lang.RuntimePermission "queuePrintJob";
 permission java.net.SocketPermission "*", "connect";
 permission java.util.PropertyPermission "*", "read";
};
```

If additional permissions are required for an application or for one or more modules of an application, use the was.policy file for that application. For example, use codeBase of ${application} and add required permissions to grant additional permissions to the entire application. Similarly, use codeBase of ${webComponent} and ${ejbComponent} to grant additional permissions to all the web modules and all the enterprise bean modules in the application. You can assign additional permissions to each module (.war file or .jar file), as shown in the following example.

This example illustrates adding extra permissions for an application in the was.policy file:

Attention: In the following code sample, the permission for the EJB module was split onto two lines for illustration only. You actually enter the permission on one line.

```
// grant additional permissions to a web module
grant codeBase " file:aWebModule.war" {
 permission java.security.SecurityPermission "printIdentity";
};

// grant additional permission to an EJB module
grant codeBase "file:aEJBModule.jar"  {
    permission java.io.FilePermission "
```

```
      ${user.install.root}${/}bin${/}DefaultDB${/}-", "read,write,delete";
   // where, ${user.install.root} is the system property whose value is
   // located in the app_server_root directory.
};
```

To use a `was.policy` file for your application, perform the following steps:

Procedure

1. Create a `was.policy` file using the policy tool. For more information on using the policy tool, see “Using PolicyTool to edit policy files for Java 2 security” on page 889.
2. Add the required permissions in the `was.policy` file using the policy tool.
3. Place the `was.policy` file in the application enterprise archive (EAR) file under the `META-INF` directory. Update the application EAR file with the newly created `was.policy` file by using the **jar** command.
4. Verify that the `was.policy` file is inserted and start an assembly tool.

   Note: An assembly tool is not available. Use an assembly tool on another platform such as Linux Intel or Windows.
5. Verify that the `was.policy` file in the application is syntactically correct. In an assembly tool, right-click the enterprise application module and click **Run Validation**.

Results

An application EAR file is now ready to run when Java 2 security is enabled.

Example

This step is required for applications to run properly when Java 2 security is enabled. If the `was.policy` file is not created and it does not contain required permissions, the application might not access system resources.

The symptom of the missing permissions is the java.security.AccessControlException exception. The missing permission is listed in the exception data, for example,

```
java.security.AccessControlException: access denied (java.io.FilePermission
${was.install.root}/java/ext/mail.jar read)
```

When an application program receives this exception and adding this permission is justified, include the permission in the `was.policy` file, for example,

```
grant codeBase "file:user_client_installed_location" {
  permission java.io.FilePermission
"${was.install.root}$(/)java$(/)jre$(/)lib$(/)ext$(/)mail.jar", "read";
};
```

The previous permission information lines are split for the illustration. Enter the permission on one line.

What to do next

Install the application.

Configuring static policy files in Java 2 security:

By configuring the static policy files, the required permission will be granted for all of the Java programs.

Before you begin

Java 2 security uses several policy files to determine the granted permission for each Java program.

See the topic about Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server.

Two types of policy files are supported by WebSphere Application Server: dynamic policy files and static policy files. Static policy files provide the default permissions. Dynamic policy files provide application permissions.

Table 98. Policy Files. This table lists the policy files.

Policy file name	Description
`java.policy`	Contains default permissions for all of the Java programs on the node. This file seldom changes.
`server.policy`	Contains default permissions for all of the WebSphere Application Server programs on the node. This file is rarely updated.
`client.policy`	Contains default permissions for all of the applets and client containers on the node.

The static policy file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not get replicated to the other machine.

Procedure

1. Identify the policy file to update.
   - If the permission is required only by an application, update the dynamic policy file. Refer to "Configuring Java 2 security policy files" on page 890.
   - If the permission is required only by applets and client containers, update the `client.policy` file. Refer to "client.policy file permissions" on page 907.
   - If the permission is required only by WebSphere Application Server (servers, agents, managers and application servers), update the `server.policy` file. Refer to "server.policy file permissions" on page 906.
   - If the permission is required by all of the Java programs running on the Java virtual machine (JVM), update the `java.policy` file. Refer to "java.policy file permissions."
2. Stop and restart WebSphere Application Server.

Results

The required permission is granted for all of the Java programs that run with the restarted JVM.

Example

If Java programs on a node require permissions, the policy file needs updating. If the Java program that required the permission is not part of an enterprise application, update the static policy file. The missing permission results in the creation of the java.security.AccessControlException exception. The missing permission is listed in the exception data.

For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:/WAS_HOME/lib/mail-impl.jar read)
```

When a Java program receives this exception and adding this permission is justified, add a permission to an adequate policy file.

For example:

```
grant codeBase "file:user_client_installed_location" {
  permission java.io.FilePermission
  "C:/WAS_HOME/lib/mail-impl.jar",
  "read";
};
```

To decide whether to add a permission, refer to Access control exception for Java 2 security.

java.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

See Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server.

The `java.policy` file is a global default policy file that is shared by all of the Java programs that run in the Java virtual machine (JVM) on the node. A change to the `java.policy` file is local for the node. The default Java policy is picked up automatically. Syntax errors in the policy files cause the application server to fail. An updated `java.policy` file is applied to all the Java programs that run in all the JVMs on the local node. Restart the programs for the updates to take effect. Modifying this file is not recommended. If a specific change is required to some of the Java programs on a node and the `java.policy` file requires updating, carefully modify the `java.policy` file with the policy tool. For more information, see "Using PolicyTool to edit policy files for Java 2 security" on page 889.

Default permissions for the `java.policy` file

The `java.policy` file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not get replicated to the other machine. The `java.policy` file that is supplied by WebSphere Application Server is located at `install_root/java/jre/lib/security/java.policy`. This file contains these default permissions.

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
        permission java.security.AllPermission;
};
// default permissions granted to all domains
grant {
        // Allows any thread to stop itself using the java.lang.Thread.stop()
        // method that takes no argument.
        // Note that this permission is granted by default only to remain
        // backwards compatible.
        // It is strongly recommended that you either remove this permission
        // from this policy file or further restrict it to code sources
        // that you specify, because Thread.stop() is potentially unsafe.
        // See "http://java.sun.com/notes" for more information.
        // permission java.lang.RuntimePermission "stopThread";

        // allows anyone to listen on un-privileged ports
        permission java.net.SocketPermission "localhost:1024-", "listen";

        // "standard" properties that can be read by anyone

        permission java.util.PropertyPermission "java.version", "read";
        permission java.util.PropertyPermission "java.vendor", "read";
        permission java.util.PropertyPermission "java.vendor.url", "read";
        permission java.util.PropertyPermission "java.class.version", "read";
        permission java.util.PropertyPermission "os.name", "read";
        permission java.util.PropertyPermission "os.version", "read";
        permission java.util.PropertyPermission "os.arch", "read";
        permission java.util.PropertyPermission "file.separator", "read";
        permission java.util.PropertyPermission "path.separator", "read";
        permission java.util.PropertyPermission "line.separator", "read";

        permission java.util.PropertyPermission "java.specification.version", "read";
        permission java.util.PropertyPermission "java.specification.vendor", "read";
        permission java.util.PropertyPermission "java.specification.name", "read";

        permission java.util.PropertyPermission "java.vm.specification.version","read";
        permission java.util.PropertyPermission "java.vm.specification.vendor","read";
        permission java.util.PropertyPermission "java.vm.specification.name", "read";
        permission java.util.PropertyPermission "java.vm.version", "read";
        permission java.util.PropertyPermission "java.vm.vendor", "read";
        permission java.util.PropertyPermission "java.vm.name", "read";
       };
```

If some Java programs on a node require permissions that are not defined as defaults in the `java.policy` file, consider updating the `java.policy` file. Most of the time, other policy files are updated instead of the `java.policy` file. The missing permission causes the creation of the , java.security.AccessControlException exception. The missing permission is listed in the exception data.

For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar read)
```

The previous two lines are one continuous line.

When a Java program receives this exception and adding this permission is justified, add a permission to the `java.policy` file.

For example:

```
grant codeBase "file:user_client_installed_location" {
permission java.io.FilePermission
"C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar", "read"; };
```

To decide whether to add a permission, refer to Access control exception for Java 2 security.

Restart all of the Java processes for the updated `java.policy` file to take effect.

server.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

See Java 2 security policy files for the list of available policy files that are supported by WebSphere Application Server.

The `server.policy` file is a default policy file that is shared by all of the WebSphere Application Servers on a node. The `server.policy` file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not replicate to the other machine.

If the default permissions for a server (the union of the permissions that is defined in the `java.policy` file and the `server.policy` file) are enough, no action is required. The default server policy is picked up automatically. If a specific change is required to some of the server programs on a node, update the `server.policy` file with the Policy Tool. Refer to the “Using PolicyTool to edit policy files for Java 2 security” on page 889 topic to edit policy files. Changes to the `server.policy` file are local for the node. Syntax errors in the policy files cause the application server to fail. Edit these policy files carefully. An updated `server.policy` file is applied to all the server programs on the local node. Restart the servers for the updates to take effect.

If you want to add permissions to an application, use the `app.policy` file and the `was.policy` file.

Note: Updates to the `app.policy` file only apply to the enterprise applications on the node to which the `app.policy` file belongs.

When you do need to modify the `server.policy` file, locate this file at: *profile_root*`/properties/server.policy`. This file contains these default permissions:

```
// Allow to use sun tools
grant codeBase "file:${java.home}/lib/tools.jar" {
permission java.security.AllPermission;
};


// Allow the WebSphere deploy tool all permissions
grant codeBase "file:${was.install.root}/deploytool/-" {
permission java.security.AllPermission;
};


grant codeBase "file:${was.install.root}/plugins/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${was.install.root}/classes/-" {
permission java.security.AllPermission;
};
```

```
grant codeBase "file:${was.install.root}/lib/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${smpe.install.root}/lib/-" {
permission java.security.AllPermission;
};

grant codeBase "file:${smpe.install.root}/-" {
permission java.security.AllPermission;
};

// Allow to use TAM
grant codeBase "file:${was.install.root}/tivoli/tam/PD.jar" {
permission java.security.AllPermission;
};
```

If some server programs on a node require permissions that are not defined as defaults in the `server.policy` file and the `server.policy` file, update the `server.policy` file. The missing permission creates the java.security.AccessControlException exception. The missing permission is listed in the exception data.

For example:

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\WebSphere\AppServer\java\jre\lib\ext\mail-impl.jar read)
```

The previous two lines are split into two lines for illustrative purposes only.

When a Java program receives this exception and adding this permission is justified, add a permission to the `server.policy` file.

For example:

```
grant codeBase "file:user_client_installed_location" {
permission java.io.FilePermission
"C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar", "read"; };
```

To decide whether to add a permission, refer to Access control exception for Java 2 security.

Restart all of the Java processes for the updated `server.policy` file to take effect.

client.policy file permissions:

Java 2 security uses several policy files to determine the granted permission for each Java program.

For the list of available policy files that are supported by WebSphere Application Server, see Java 2 security policy files.

- The `client.policy` file is a default policy file that is shared by all of the WebSphere Application Server client containers and applets on a node.
- The union of the permissions that is contained in the `java.policy` file and the `client.policy` file are given to all of the client containers for WebSphere Application Server and applets running on the node.
- The `client.policy` file is not a configuration file that is managed by the repository and the file replication service. Changes to this file are local and do not replicate to the other machine.
- The `client.policy` file supplied by WebSphere Application Server is located in the *profile_root*`/properties/client.policy`.
- If the default permissions for a client (union of the permissions defined in the `java.policy` file and the `client.policy` file) are enough, no action is required. The default client policy is picked up automatically.

- If a specific change is required to some of the client containers and applets on a node, modify the `client.policy` file with the Policy Tool. Refer to “Using PolicyTool to edit policy files for Java 2 security” on page 889, to edit policy files. Changes to the `client.policy` file are local for the node.

This file contains these default permissions:

```
grant codeBase "file:${was.install.root}/java/ext/*" {
  permission java.security.AllPermission;
};

// JDK classes
grant codeBase "file:${was.install.root}/java/ext/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/java/tools/ibmtools.jar" {
  permission java.security.AllPermission;
};
grant codeBase "file:/QIBM/ProdData/Java400/jdk14/lib/tools.jar" {
  permission java.security.AllPermission;
};

// WebSphere system classes
grant codeBase "file:${was.install.root}/lib/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/plugins/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/classes/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${was.install.root}/installedConnectors/-" {
  permission java.security.AllPermission;
};
grant codeBase "file:${user.install.root}/installedConnectors/-" {
  permission java.security.AllPermission;
};

grant codeBase "file:${was.install.root}/installedChannels/-" {
  permission java.security.AllPermission;
};

// J2EE 1.4 permissions for client container applications
// in $WAS_HOME/installedApps
grant codeBase "file:${user.install.root}/installedApps/-" {
  //Application client permissions
  permission java.awt.AWTPermission "accessClipboard";
  permission java.awt.AWTPermission "accessEventQueue";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission java.lang.RuntimePermission "exitVM";
  permission java.lang.RuntimePermission "loadLibrary";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
  permission java.net.SocketPermission "localhost:1024-", "accept,listen";
  permission java.io.FilePermission "*", "read,write";
  permission java.util.PropertyPermission "*", "read";

};

// J2EE 1.4 permissions for client container - expanded ear file code base
grant codeBase "file:${com.ibm.websphere.client.applicationclient.archivedir}/-" {
  permission java.awt.AWTPermission "accessClipboard";
  permission java.awt.AWTPermission "accessEventQueue";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission java.lang.RuntimePermission "exitVM";
  permission java.lang.RuntimePermission "loadLibrary";
  permission java.lang.RuntimePermission "queuePrintJob";
  permission java.net.SocketPermission "*", "connect";
```

```
  permission java.net.SocketPermission "localhost:1024-", "accept,listen";
  permission java.io.FilePermission "*", "read,write";
  permission java.util.PropertyPermission "*", "read";
};
```

All of the client containers and applets on the local node are granted the updated permissions when they start. If some client containers or applets on a node require permissions that are not defined as defaults in the `java.policy` file and the default `client.policy` file, update the `client.policy` file. The missing permission creates the java.security.AccessControlException exception. The missing permission is listed in the exception data, for example,

```
java.security.AccessControlException: access denied (java.io.FilePermission
C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar read)
```

The previous two lines of the example are one continuous line, but presented as such for illustrative purposes only.

When a client program receives this exception and adding this permission is justified, add a permission to the `client.policy` file, for example:

```
grant codebase "file:user_client_installed_location" {permission
 java.io.FilePermission "C:\WebSphere\AppServer\java\jre\lib\ext\mail.jar", "read"; };
```

To decide whether to add a permission, refer to Access control exception for Java 2 security.

If you update the policy file, you must restart the browser and any client applications.

Developing with programmatic security APIs for web applications

Use this information to programmatically secure APIs for web applications.

Before you begin

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application.

The authenticate, login, logout, getRemoteUser, isUserInRole and getAuthType servlet security methods are methods of the javax.servlet.http.HttpServletRequest interface. For more detailed information about these servlet security methods, read the Servlet security methods article.

Note:

> The logout, login, and authenticate APIs are new for Java Servlet 3.0 in this release of WebSphere Application Server.

You can configure several options for web authentication that determine how the web client interacts with protected and unprotected Uniform Resource Identifiers (URI). Also, you can specify whether WebSphere Application Server challenges the web client for basic authentication information if the certificate authentication for the HTTPS client fails. For more information, see the Selecting an authentication mechanism article.

You can enable a login module to indicate which principal class is returned by these calls. Refer to the topic about mapping a registry principal to a System Authorization Facility user ID using a Java Authentication and Authorization Services login module for more information.

When the isUserInRole method is used, declare a security-role-ref element in the deployment descriptor with a role-name subelement containing the role name that is passed to this method, or use the @DeclareRoles annotation. Because actual roles are created during the assembly stage of the application, you can use a logical role as the role name and provide enough hints to the assembler in the description of the security-role-ref element to link that role to the actual role. During assembly, the assembler creates

a role-link subelement to link the role name to the actual role. Creation of a security-role-ref element is possible if an assembly tool, such as Rational Application Developer, is used. You also can create the security-role-ref element during assembly stage using an assembly tool.

Procedure

1. Add the required security methods in the servlet code.
2. Create a security-role-ref element with the **role-name** field. If a security-role-ref element is not created during development, make sure it is created during the assembly stage.

Results

A programmatically secured servlet application.

Example

These steps are required to secure an application programmatically. This action is particularly useful when a web application needs to access external resources and wants to control the access to external resources using its own authorization table (external-resource to remote-user mapping). In this case, use the getUserPrincipal or the getRemoteUser methods to get the remote user, then the application can consult its own authorization table to perform authorization. The remote user information also can help retrieve the corresponding user information from an external source such as a database or from an enterprise bean. You can use the isUserInRole method in a similar way.

After development, you can create a security-role-ref element:

```
<security-role-ref>
   <description>Provide hints to assembler for linking this role
                name to an actual role here<\description>
   <role-name>Mgr<\role-name>
</security-role-ref>
```

During assembly, the assembler creates a role-link element:

```
<security-role-ref>
   <description>Hints provided by developer to map the role
               name to the role-link</description>
   <role-name>Mgr</role-name>
   <role-link>Manager</role-link>
</security-role-ref>
```

You can add programmatic servlet security methods inside any servlet doGet, doPost, doPut, and doDelete service methods. The following example depicts using a programmatic security API:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response) {

   ....
   // to logoff the current user
  request.logout();

   // to login with a new user
   request.login("bob","bobpwd")

   // to get remote user using getUserPrincipal()
   java.security.Principal principal = request.getUserPrincipal();
   String remoteUser = principal.getName();

   // to get remote user using getRemoteUser()
   remoteUser = request.getRemoteUser();

   // to check if remote user is granted Mgr role
   boolean isMgr = request.isUserInRole("Mgr");

   // use the above information in any way as needed by
   // the application
   ....

}
```

You can programmatic login with a user ID and password inside any servlet doGet, doPost, doPut, and doDelete service methods. The following example depicts using a programmatic login/logout API:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response) {

   ....
   // to logout the current user. If you are not already authenticate, then no need to call the logout() method.
   request.logout();

   // to login with a new user
   request.login("utle","mypwd")

   // the user utle subject now set on the thread and the LTPA SSO cookie is set in the response
   ....

}
```

You can programmatic authenticate with a different identity inside any servlet doGet, doPost, doPut, and doDelete service methods. In this example, if the web servlet is configured to use basicAuth, the web server returns a response code 401, the login prompt is displayed, and you can enter the user ID and password to authenticate. The following example depicts using a programmatic login/logout API:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response) {

   ....
   // to logout the current user. If you are not already authenticate, then no need to call the logout() method.

   // to login with a new user
   request.authenticate(response);

   // the new user subject now set on the thread and the LTPA SSO cookie is set in the response
   ....

}
```

When developing Servlet 3.0 modules, the value of the rolename argument in isCallerInRole method can be defined using Java annotations instead of declaring a security-role-ref elements in the deployment descriptor.

```
@javax.annotation.security.DeclareRoles("Mgr")
public void doGet(HttpServletRequest request,
HttpServletResponse response) {

   ....

   // to get remote user using getUserPrincipal()
   java.security.Principal principal = request.getUserPrincipal();
   String remoteUser = principal.getName();

   // to get remote user using getRemoteUser()
   remoteUser = request.getRemoteUser();

   // to check if remote user is granted Mgr role
   boolean isMgr = request.isUserInRole("Mgr");

   // use the above information in any way as needed by
   // the application
   ....

}
```

The following example depicts a web application or servlet using the programmatic security model.

This example illustrates one use and not necessarily the only use of the programmatic security model. The application can use the information that is returned by the getUserPrincipal, isUserInRole, and the getRemoteUser methods in any other way that is meaningful to that application. Use the declarative security model whenever possible.

File : HelloServlet.java

```
public class HelloServlet extends javax.servlet.http.HttpServlet {

 public void doPost(
  javax.servlet.http.HttpServletRequest request,
  javax.servlet.http.HttpServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException {
 }
public void doGet(
```

```
  javax.servlet.http.HttpServletRequest request,
  javax.servlet.http.HttpServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException {

        String s = "Hello";


        // get remote user using getUserPrincipal()
        java.security.Principal principal = request.getUserPrincipal();
        String remoteUserName = "";
        if( principal != null )
          remoteUserName = principal.getName();
// get remote user using getRemoteUser()
        String remoteUser = request.getRemoteUser();

        // check if remote user is granted Mgr role
        boolean isMgr = request.isUserInRole("Mgr");

        // display Hello username for managers and bob.
        if ( isMgr || remoteUserName.equals("bob") )
            s = "Hello " + remoteUserName;

   String message =  "<html> \n" +
              "<head><title>Hello Servlet</title></head>\n" +
         "<body> /n +"
    "<h1> "  +s+ </h1>/n " +
  byte[] bytes = message.getBytes();

  // displays "Hello" for ordinary users
        // and displays "Hello username" for managers and "bob".
        response.getOutputStream().write(bytes);
 }

}
```

After developing the servlet, you can create a security role reference for the HelloServlet servlet as shown in the following example:

```
<security-role-ref>
     <description> </description>
     <role-name>Mgr</role-name>
</security-role-ref>
```

What to do next

After developing an application, use an assembly tool to create roles and to link the actual roles to role names in the security-role-ref elements. See the information about securing web applications using an assembly tool.

Servlet security methods:

The authenticate, login, logout, getRemoteUser, isUserInRole and getAuthType servlet security methods are methods of the javax.servlet.http.HttpServletRequest interface.

authenticate

Note: The authenticate, login and logout servlet security methods are new for Java Servlet 3.0 in this release of WebSphere Application Server.

The authenticate method authenticates a user by using the WebSphere Application Server container login mechanism configured for the servlet context.

The syntax of the authenticate method is as follows:

```
boolean authenticate(HttpServletResponse response))
```

The previous example uses the following element:

response
The HttpServletResponse associated with the HttpServletRequest.

The authenticate method returns `true` when authentication has been established or authentication is successful.

The authenticate method returns `false` if authentication is incomplete and the underlying login mechanism has committed, in the response, the message and HTTP status code to be returned to the user.

A java.io.IOException occurs if an error occurs while writing the response.

A ServletException occurs if the authentication failed, and the caller is responsible for handling the error (for example, the underlying login mechanism did not establish the message and the HTTP status code to be returned to the user).

Note: When the authenticate method is called, be aware of the following:

- WebSphere Application Server returns `HTTP 401` code to a client.
- The method depends on the WebSphere Application Server container login mechanism that is configured for the servlet context. For example, if you have a form login defined for this servlet, it prompts for a user name and password. The client sends the user ID and password to WebSphere Application Server for authentication.

Important: Make sure that the authenticate method returns `true` before using the new subject to call another service. For example:

```
Boolean authResultTrue = req.authenticate(response);
  if (!authResultTrue) {
 return;
} else {
 // Use the new invocation subject to call other services.
       }
```

login

The login method authenticates a user to the WebSphere Application Server with a user ID and password. If authentication is successful, it creates a user subject on the thread and Lightweight Third Party Authentication (LTPA) cookies (if single sign-on (SSO) is enabled).

The syntax of the login method is as follows:

```
login(java.lang.String username, java.lang.String password)
```

The previous example uses the following elements:

username
The string value that corresponds to the login identifier of the user.

password
The password of the user.

A ServletException occurs if the configured login mechanism does not support username and password authentication, if an identity had already been authenticated (prior to the call to login), or if validation of the provided username and password fails.

Note: You can set the security custom property com.ibm.websphere.security.webAlwaysLogin to `true` and it will authenticate to the WebSphere application with the username and password, even if it is already authenticated.

For more information about modifying security custom properties, read the Modifying an existing custom property in a global security configuration or in a security domain configuration article.

Note: The login method always uses the user ID and password to authenticate to the WebSphere application server and even the SSO information that is present in the HttpServletRequest.

Note: The authenticate and login methods set the invocation subject to the new subject. If the caller subject is null, it then sets the caller subject to the new subject. If the caller subject is not null, then the caller subject is not set to the new subject.

Since the authenticate and login methods set the invocation subject to the new subject, the RunAs defined by the run-As attribute in deployment descriptor, security annotation or dynamic annotation is ignored.

logout

The logout method logs the user out of the WebSphere Application Server and invalidates the HTTP session. During this process, WebSphere Application Server completes the following processes:

- Clears the LTPA cookies if SSO is enabled
- Invalidates the HTTP session
- Removes the user from the authentication cache
- Removes the user subject from the thread
- Clears the caller and invocation subjects
- Sets the authentication type to `null`

After logging out, access to a protected web resource requires re-authentication and the getUserPrincipal, getRemoteUser and getAuthType methods return `null`.

The syntax of the logout method is as follows:

```
logout()
```

A ServletException occurs if the logout fails.

Audit event types for the authenticate, login and logout methods

To audit authenticate, login and logout methods, you must create or extend some audit event type files. These event type are not part of the default event type files.

Table 99. Audit event types for the authenticate, login, and logout methods.

The audit event types required for the authenticate, login, and logout methods are:

Method	Audit event name	Audit outcome of the event
authenticate/login	SECURITY_AUTHN	SUCCESS and or FAILURE
logout	SECURITY_AUTHN_TERMINATE	SUCCESS
logout	SECURITY_AUTHN_TERMINATE	FAILURE

isUserInRole

(String role name): Returns `true` if the remote user is granted the specified security role. If the remote user is not granted the specified role, or if no user is authenticated, it returns `false`.

getRemoteUser

The getRemoteUser method returns the login of the user that makes the request if the user has been authenticated. If the user has not been authenticated, the getRemoteUser method returns `null`.

getAuthType

The getAuthType method returns the name of the authentication scheme that is used to protect the servlet. If the servlet is not protected, the getAuthType method returns `null`.

The authentication schemes used are:

FORM when form-based authentication is used

BASIC
when basic authentication is used.

CLIENT_CERT
when client certificate authentication is used.

Note:

For both the getRemoteUser and getAuthType methods, the data that is returned depends upon whether security is enabled in the application server where the servlet is deployed. The following possibilities exist:

- If application security is enabled and a servlet is protected, then the getRemoteUser method returns the login and the getAuthType method returns the configured authentication scheme.
- If application security is not enabled, both methods return `null`.

Web authentication settings:

Use this page to specify the web authentication settings that are associated with a web client.

To view this administrative console page, complete the following steps:

1. Click **Security** > **Global security**.
2. Under Authentication, expand `Web and SIP security` and click `General settings`.

You can override the global Web authentication settings that you select on this panel by specifying one or more of the following system properties for the controller and the servant. Complete the following steps to specify one of these system properties for the controller:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Server infrastructure, click **Java and Process Management** > **Process definition**.
3. Under Additional properties, click **Java Virtual Machine** > **Custom properties** > **New**

Complete the following steps to specify one of these system properties for the servant:

1. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name***.
2. Under Server infrastructure, click **Java and Process Management** > **Process definition**.
3. Under Additional properties, click **Java Virtual Machine** > **Custom properties** > **New**

Table 100. Web authentication system property values. This table lists the web authentication system property values.

Property name	Value	Explanation
com.ibm.wsspi.security.web.webAuthReq	lazy	This value is equivalent to the `Authenticate only when the URI is protected` option. **Note:** You can set webAuthReq differently through the administrative console or scripting when using a global or a security domain, but the global level always takes precedence.
com.ibm.wsspi.security.web.webAuthReq	persisting	This value is equivalent to the `Use available authentication data when an unprotected URI is accessed` option.
com.ibm.wsspi.security.web.webAuthReq	always	This value is equivalent to the `Authenticate when any URI is accessed` option.
com.ibm.wsspi.security.web.failOverToBasicAuth	true	This value is equivalent to the `Default to basic authentication when certificate authentication for the HTTPS client fails` option.

Authenticate only when the URI is protected:

The application server challenges the web client to provide authentication data when the web client accesses a Uniform Resource Identifier (URI) that is protected by a Java 2 Platform, Enterprise Edition (J2EE) role. The authenticated identity is available only when the web client accesses a protected URI.

This option is the default J2EE web authentication behavior that is also available in previous releases of WebSphere Application Server.

Note: When you select this option, the administrative console login page is missing images. You might encounter the following error in the administrative console: “CWLAA6003: Could not display the portlet, the portlet may not be started. Check the error logs”.

The missing images and the error message are a side-effect of this option. The images do not display because the URIs for the images now need authentication, which requires you to log in. You can ignore this error message.

Information	**Value**
Default:	Enabled

Use available authentication data when an unprotected URI is accessed:

The web client can access validated authenticated data that it previously could not access. This option enables the web client to call the getRemoteUser, isUserInRole, and getUserPrincipal methods to retrieve an authenticated identity from an unprotected URI.

When you select this option with the `Authenticate only when the URI is protected` option, the web client can use authenticated data when the URI is protected or not protected.

When this option is selected and Form-based authentication is being used, a WASPostParam cookie is generated during the authentication procedure of the HTTP POST request even if the target URL is unprotected. A WASPOSTParam cookie is a temporary cookie used to store HTTP POST parameters. This results in the Web client being sent the unnecessary cookie with an HTTP response. This might cause unexpected behavior when the size of the cookie is larger than the browser limit. To avoid this behavior, a custom property, com.ibm.websphere.security.util.postParamMaxCookieSize can be set to cause the security code to stop generating the cookie if the maximum size is reached.

Important: This option does not challenge the web client to provide authenticated data if the web client accesses an unprotected URI without authenticated data.

Information	**Value**
Default:	Enabled

Authenticate when any URI is accessed:

The web client must provide authentication data regardless of whether the URI is protected.

Information	**Value**
Default:	Disabled

Default to basic authentication when certificate authentication for the HTTPS client fails:

When the required HTTPS client certificate authentication fails, the application server uses the basic authentication method to challenge the web client to provide a user ID and password.

The HTTP client certification authentication that is performed by the application server security is different from the client authentication that is performed by the web server plug-in. If you configure the web server plug-in for mutual authentication and client authentication fails, the following situations will occur:

- The web server produces a error and the web request is not processed by application server security.
- The application server cannot fail over to basic authentication.

Information	Value
Default:	Disabled

Developing with programmatic APIs for EJB applications

Use this topic to programmatically secure your Enterprise JavaBeans (EJB) applications.

About this task

Programmatic security is used by security-aware applications when declarative security alone is not sufficient to express the security model of the application. The javax.ejb.EJBContext application programming interface (API) provides two methods whereby the bean provider can access security information about the enterprise bean caller.

- **IsCallerInRole**(String rolename): Returns `true` if the bean caller is granted the security role that is specified by role name. If the caller is not granted the specified role, or if the caller is not authenticated, it returns `false`. If the specified role is granted Everyone access, it always returns `true`.
- **getCallerPrincipal**: Returns the java.security. Principal object that contains the bean caller name. If the caller is not authenticated, it returns a principal that contains an unauthorized name.

You can enable a login module to indicate which principal class is returned by these calls.

When the isCallerInRole method is used, declare a security-role-ref element in the deployment descriptor with a role-name that is subelement containing the role name that is passed to this method. Because actual roles are created during the assembly stage of the application, you can use a logical role as the role name and provide enough hints to the assembler in the description of the security-role-ref element to link that role to an actual role. During assembly, the assembler creates a role-link subelement to link the role-name to the actual role. Creation of a security-role-ref element is possible if an assembly tool such as Rational Application Developer is used. You also can create the security-role-ref element during the assembly stage using an assembly tool.

Procedure

1. Add the required security methods in the EJB module code.
2. Create a security-role-ref element with a role-name field for all the role names used in the isCallerInRole method. If a security-role-ref element is not created during development, make sure it is created during the assembly stage.

Results

Performing the previous steps result in a programmatically secured EJB application.

Example

Hard coding security policies in applications is strongly discouraged. The Java Platform, Enterprise Edition (Java EE) security model capabilities of declaratively specifying security policies is encouraged wherever possible. Use these APIs to develop security-aware EJB applications.

Using Java EE security model capabilities to specify security policies declaratively is useful when an EJB application wants to access external resources and wants to control the access to these external resources using its own authorization table (external-resource to user mapping). In this case, use the

getCallerPrincipal method to get the caller identity and then the application can consult its own authorization table to perform authorization. The caller identification also can help retrieve the corresponding user information from an external source, such as database or from another enterprise bean. You can use the isCallerInRole method in a similar way.

After development, you can create a security-role-ref element:

```
<security-role-ref>
<description>Provide hints to assembler for linking this role-name to
actual role here<\description>
<role-name>Mgr<\role-name>
</security-role-ref>
```

During assembly, the assembler creates a role-link element:

```
<security-role-ref>
<description>Hints provided by developer to map role-name to role-link</description>
<role-name>Mgr</role-name>
<role-link>Manager</role-link>
</security-role-ref>
```

You can add programmatic EJB component security methods for example isCallerInRole and getCallerPrincipal, inside any business methods of an enterprise bean. The following example of programmatic security APIs includes a session bean:

```
public class aSessionBean implements SessionBean {

       .....

       // SessionContext extends EJBContext. If it is entity bean use EntityContext
       javax.ejb.SessionContext context;

       // The following method will be called by the EJB container
       // automatically
       public void setSessionContext(javax.ejb.SessionContext ctx) {
              context = ctx; // save the session bean's context
       }

       ....

       private  void aBusinessMethod()  {
       ....

       // to get  bean's caller using getCallerPrincipal()
       java.security.Principal principal = context.getCallerPrincipal();
       String  callerId= principal.getName();

       // to check if  bean's caller is granted Mgr role
       boolean isMgr = context.isCallerInRole("Mgr");

       // use the above information in any way as needed by the
       //application

       ....
       }

       ....
}
```

When developing EJB 3.x modules, the value of the rolename argument in isCallerInRole method can be defined using Java annotations instead of declaring a security-role-ref elements in the deployment descriptor.

```
@javax.annotation.security.DeclareRoles("Mgr")
@Stateless           // annotation is used to indicate a session bean
public class aSessionBean implements MyBusinessInterface { //you don't have to extend sessionbean interface
```

```
        .....
        // SessionContext extends EJBContext. In EJB 3.0 use Resource annotation to inject context
    @Resource
        javax.ejb.SessionContext context;       }

       ....

       private  void aBusinessMethod()  {
       ....

       // to get  bean's caller using getCallerPrincipal()
       java.security.Principal principal = context.getCallerPrincipal();
       String  callerId= principal.getName();

       // to check if  bean's caller is granted Mgr role
       boolean isMgr = context.isCallerInRole("Mgr");

       // use the above information in any way as needed by the
       //application

       ....
       }

       ....
}
```

What to do next

After developing an application, use an assembly tool to create roles and to link the actual roles to role names in the security-role-ref elements. See the information about securing web applications using an assembly tool.

Example: Enterprise bean application code:

The following Enterprise JavaBeans (EJB) component example illustrates the use of the isCallerInRole and the getCallerPrincipal methods in an EJB module.

Using declarative security is recommended. The following example is one way of using the isCallerInRole and the getCallerPrincipal methods. The application can use this result in any way that is suitable.

A remote interface

```
File : Hello.java

package tests;
import java.rmi.RemoteException;
/**
 * Remote interface for Enterprise Bean: Hello
 */
public interface Hello extends javax.ejb.EJBObject {
      public abstract String getMessage()throws RemoteException;
     public abstract void setMessage(String s)throws RemoteException;
}
```

A home interface

```
File : HelloHome.java
package tests;
/**
 * Home interface for Enterprise Bean: Hello
 */
public interface HelloHome extends javax.ejb.EJBHome {
 /**
  * Creates a default instance of Session Bean: Hello
  */
 public tests.Hello create() throws javax.ejb.CreateException,
  java.rmi.RemoteException;
}
```

A bean implementation

File : HelloBean.java

```
package tests;
/**
 * Bean implementation class for Enterprise Bean: Hello
 */
public class HelloBean implements javax.ejb.SessionBean {
 private javax.ejb.SessionContext mySessionCtx;
 /**
  * getSessionContext
  */
 public javax.ejb.SessionContext getSessionContext() {
  return mySessionCtx;
 }
 /**
  * setSessionContext
  */
 public void setSessionContext(javax.ejb.SessionContext ctx) {
  mySessionCtx = ctx;
 }
 /**
  * ejbActivate
  */
 public void ejbActivate() {
 }
 /**
  * ejbCreate
  */
 public void ejbCreate() throws javax.ejb.CreateException {
 }
 /**
  * ejbPassivate
  */
 public void ejbPassivate() {
 }
 /**
  * ejbRemove
  */
 public void ejbRemove() {
 }

 public java.lang.String message;


      //business methods

      // all users can call getMessage()
      public String getMessage() {
          return message;
      }

      // all users can call setMessage() but only few users can set new message.
      public void setMessage(String s) {

        // get  bean's caller using getCallerPrincipal()
        java.security.Principal principal = mySessionCtx.getCallerPrincipal();
        java.lang.String  callerId= principal.getName();

        // check if  bean's caller is granted Mgr role
        boolean isMgr = mySessionCtx.isCallerInRole("Mgr");

        // only set supplied message if caller is "bob" or caller is granted Mgr role
        if ( isMgr || callerId.equals("bob") )
            message = s;
        else
            message = "Hello";
      }

}
```

After the development of the entity bean, create a security role reference in the deployment descriptor under the session bean, Hello:

```
<security-role-ref>
     <description>Only Managers can call setMessage() on this bean (Hello)</description>
     <role-name>Mgr</role-name>
</security-role-ref>
```

For an explanation of how to create a <security-role-ref> element, see Securing enterprise bean applications. Use the information under Map security-role-ref and role-name to role-link to create the element.

Customizing web application login

You can create a form login page and an error page to authenticate a user.

Before you begin

A web client or a browser can authenticate a user to a Web server using one of the following mechanisms:

- **HTTP basic authentication**: A web server requests the Web client to authenticate and the web client passes a user ID and a password in the HTTP header.
- **HTTPS client authentication**: This mechanism requires a user (web client) to possess a public key certificate. The web client sends the certificate to a web server that requests the client certificates. This authentication mechanism is strong and uses the Hypertext Transfer Protocol with Secure Sockets Layer (HTTPS) protocol.
- **Form-based Authentication**: A developer controls the look and feel of the login screens using this authentication mechanism.

The Hypertext Transfer Protocol (HTTP) basic authentication transmits a user password from the web client to the web server in simple base64 encoding. Form-based authentication transmits a user password from the browser to the web server in plain text. Therefore, both HTTP basic authentication and form-based authentication are not very secure unless the HTTPS protocol is used.

The web application deployment descriptor contains information about which authentication mechanism to use. When form-based authentication is used, the deployment descriptor also contains entries for login and error pages. A login page can be either an HTML page or a JavaServer Pages (JSP) file. This login page is displayed on the web client side when a secured resource (servlet, JSP file, HTML page) is accessed from the application. On authentication failure, an error page is displayed. You can write login and error pages to suit the application needs and control the look and feel of these pages. During assembly of the application, an assembler can set the authentication mechanism for the application and set the login and error pages in the deployment descriptor.

Form login uses the servlet sendRedirect method, which has several implications for the user. The sendRedirect method is used twice during form login:

- The sendRedirect method initially displays the form login page in the web browser. It later redirects the web browser back to the originally requested protected page. The sendRedirect(String URL) method tells the web browser to use the HTTP GET request to get the page that is specified in the web address. If HTTP POST is the first request to a protected servlet or JavaServer Pages (JSP) file, and no previous authentication or login occurred, then HTTP POST is not delivered to the requested page. However, HTTP GET is delivered because form login uses the sendRedirect method, which behaves as an HTTP GET request that tries to display a requested page after a login occurs.
- Using HTTP POST, you might experience a scenario where an unprotected HTML form collects data from users and then posts this data to protected servlets or JSP files for processing, but the users are not logged in for the resource. To avoid this scenario, structure your web application or permissions so that users are forced to use a form login page before the application performs any HTTP POST actions to protected servlets or JSP files.

Procedure

1. Create a form login page with the required look and feel, including the required elements to perform form-based authentication.
2. Create an error page. You can program error pages to retry authentication or to display an appropriate error message.
3. Place the login page and error page in the web application archive (`.war`) file relative to the top directory. For example, if the login page is configured as `/login.html` in the deployment descriptor, place it in the top directory of the WAR file. An assembler can also perform this step using the assembly tool.
4. Create a form logout page and insert it to the application only when the web application requires a form-based authentication mechanism.

   By default the URL to the logout page should point to the host to which the request was made or its domain. Otherwise, a generic logout page is displayed. If you need to point this URL to a different host, then you need to set the com.ibm.websphere.security.logoutExitPageDomainList property in the `security.xml` file with a list of URLs that are allowed for the logout page. You can choose to allow any logout exit page to be used by setting the com.ibm.websphere.security.allowAnyLogoutExitPageHost property to a value of `true`. Setting this property to `true` might open your systems to a potential URL redirect attacks.

Example: Form login

You can use the WebSphere Application Server login facilities to implement and configure form login procedures. Use the following technologies for WebSphere Application Server and Java Platform, Enterprise Edition (Java EE) login functionality:

- Java EE form-based login
- Java EE servlet filter with login
- IBM extension: form-based login

The form login sample is part of the Technology Samples package. For more information on how to access the form login sample, see Accessing the samples.

Form login usage

For the authentication to proceed appropriately, the action of the login form must always have the j_security_check action. The following example shows how to code the form into the HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="text" name="j_password">
<\form>
```

Use the **j_username** input field to get the user name, and use the **j_password** input field to get the user password.

On receiving a request from a web client, the web server sends the configured form page to the client and preserves the original request. When the web server receives the completed form page from the web client, the server extracts the user name and password from the form and authenticates the user. On successful authentication, the web server redirects the call to the original request. If authentication fails, the web server redirects the call to the configured error page.

The following example depicts a login page in HTML (`login.html`):

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<META HTTP-EQUIV = "Pragma" CONTENT="no-cache">
<title> Security FVT Login Page </title>
<body>
<h2>Form Login</h2>
<FORM METHOD=POST ACTION="j_security_check">
<p>
<font size="2"> <strong> Enter user ID and password: </strong></font>
```

```
<BR>
<strong> User ID</strong> <input type="text" size="20" name="j_username">
<strong> Password </strong>  <input type="password" size="20" name="j_password">
<BR>
<BR>
<font size="2">  <strong> And then click this button: </strong></font>
<input type="submit" name="login" value="Login">
</p>

</form>
</body>
</html>
```

The following example depicts an error page in a JSP file:

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<head><title>A Form login authentication failure occurred</head></title>
<body>
<H1><B>A Form login authentication failure occurred</H1></B>
<P>Authentication may fail for one of many reasons. Some possibilities include:
<OL>
<LI>The user-id or password may be entered incorrectly; either misspelled or the
wrong case was used.
<LI>The user-id or password does not exist, has expired, or has been disabled.
</OL>
</P>
</body>
</html>
```

After an assembler configures the web application to use form-based authentication, the deployment descriptor contains the login configuration as shown:

```
<login-config id="LoginConfig_1">
<auth-method>FORM<auth-method>
<realm-name>Example Form-Based Authentication Area</realm-name>
<form-login-config id="FormLoginConfig_1">
<form-login-page>/login.html</form-login-page>
<form-error-page>/error.jsp</form-error-page>
</form-login-config>
</login-config>
```

A sample web application archive (WAR) file directory structure that shows login and error pages for the previous login configuration follows:

```
META-INF
     META-INF/MANIFEST.MF
     login.html
     error.jsp
     WEB-INF/
     WEB-INF/classes/
     WEB-INF/classes/aServlet.class
```

Form logout

Form logout is a mechanism to log out without having to close all Web-browser sessions. After logging out of the form logout mechanism, access to a protected web resource requires re-authentication. This feature is not required by J2EE specifications, but it is provided as an additional feature in WebSphere Application Server security.

Suppose that you want to log out after logging into a web application and perform some actions. A form logout works in the following manner:

1. The logout-form URI is specified in the web browser and loads the form.
2. The user clicks **Submit** on the form to log out.
3. The WebSphere Application Server security code logs the user out. During this process, the Application Server completes the following processes:
   a. Clears the Lightweight Third Party Authentication (LTPA) / single sign-on (SSO) cookies
   b. Invalidates the HTTP session
   c. Removes the user from the authentication cache
4. Upon logout, the user is redirected to a logout exit page.

Form logout does not require any attributes in a deployment descriptor. The form-logout page is an HTML or a JavaServer Pages (JSP) file that is included with the web application. The form-logout page is like

most HTML forms except that like the form-login page, the form-logout page has a special post action. This post action is recognized by the web container, which dispatches the post action to a special internal form-logout servlet. The post action in the form-logout page must be `ibm_security_logout`.

You can specify a logout-exit page in the logout form and the exit page can represent an HTML or a JSP file within the same web application to which the user is redirected after logging out. Additionally, the logout-exit page permits a fully qualified URL in the form of `http://hostname:port/URL`. The logout-exit page is specified as a parameter in the form-logout page. If no logout-exit page is specified, a default logout HTML message is returned to the user.

Here is a sample form logout HTML form. This form configures the logout-exit page to redirect the user back to the login page after logout.

```
<!DOCTYPE HTML PUBliC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
  <META HTTP-EQUIV = "Pragma" CONTENT="no-cache">
  <title>Logout Page </title>
  <body>
  <h2>Sample Form Logout</h2>
      <FORM METHOD=POST ACTION="ibm_security_logout" NAME="logout">
      <p>
      <BR>
      <BR>
      <font size="2"><strong> Click this button to log out: </strong></font>
      <input type="submit" name="logout" value="Logout">
      <INPUT TYPE="HIDDEN" name="logoutExitPage" VALUE="/login.html">
      </p>
      </form>
  </body>
</html>
```

What to do next

After developing login and error pages, add them to the Web application. Use the assembly tool to configure an authentication mechanism and insert the developed login page and error page in the deployment descriptor of the application.

Developing servlet filters for form login processing

You can control the look and feel of the login screen using the form-based login mechanism. In form-based login, you specify a login page that is used to retrieve the user ID and password information. You also can specify an error page that displays when authentication fails.

About this task

If additional authentication or additional processing is required before and after authentication, servlet filters are an option. Servlet filters can dynamically intercept requests and responses to transform or to use the information that is contained in the requests or responses. One or more servlet filters can be attached to a servlet or to a group of servlets. Servlet filters also can attach to JavaServer Pages (JSP) files and HTML pages. All of the attached servlet filters are called before the servlet is invoked.

Both form-based login and servlet filters are supported by any servlet Version 2.3 specification-complaint web container. The form login servlet performs the authentication and servlet filters perform additional authentication, auditing, or logging information.

To perform pre-login and post-login actions using servlet filters, configure these filters for either form login page support or for the /j_security_check URL. The j_security_check is posted by a form login page with the j_username parameter that contains the user name and the j_password parameter that contains the password. A servlet filter can use the user name parameter and password information to perform more authentication or other special needs.

Procedure

1. A servlet filter implements the javax.servlet.Filter class. Implement three methods in the filter class:

- **init(javax.servlet.FilterConfig cfg)**. This method is called by the container once, when the servlet filter is placed into service. The FilterConfig passed to this method contains the init-parameters of the servlet filter. Specify the init-parameters for a servlet filter during configuration using the assembly tool.
- **destroy**. This method is called by the container when the servlet filter is taken out of a service.
- **doFilter(ServletRequest req, ServletResponse res, FilterChain chain)**. This method is called by the container for every servlet request that maps to this filter before invoking the servlet. The FilterChain chain that is passed to this method can be used to invoke the next filter in the chain of filters. The original requested servlet runs when the last filter in the chain calls the chain.doFilter method. Therefore, all filters call the chain.doFilter method for the original servlet to run after filtering. If an additional authentication check is implemented in the filter code and results in failure, the original servlet does not run. The chain.doFilter method is not called and can be redirected to some other error page.

2. If a servlet maps to many servlet filters, servlet filters are called in the order that is listed in the `web.xml` deployment descriptor of the application. Place the servlet filter class file in the `WEB-INF/classes` directory of the application.

Example

An example of a servlet filter.

This login filter can map to the /j_security_check URL to perform pre-login and post-login actions.

```
import javax.servlet.*;
     public class LoginFilter implements Filter {
     protected FilterConfig filterConfig;
     // Called once when this filter is instantiated.
     // If mapped to j_security_check, called
     // very first time j_security_check is invoked.
     public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
        }
     public void destroy() {
        this.filterConfig = null;
        }
      // Called for every request that is mapped to this filter.
     // If mapped to j_security_check,
     // called for every  j_security_check action
     public void doFilter(ServletRequest request,
     ServletResponse response, FilterChain chain)
         throws java.io.IOException, ServletException   {
         // perform pre-login action here
         chain.doFilter(request, response);
         // calls the next filter in chain.
         // j_security_check if this filter is
         // mapped to j_security_check.
        // perform post-login action here.
                 }
       }
```

Using servlet filters to perform pre-login and post-login processing during form login

This example illustrates one way that the servlet filters can perform pre-login and post-login processing during form login.

```
Servlet filter source code: LoginFilter.java
/**
 * A servlet filter example: This example filters j_security_check and
 * performs pre-login action to determine if the user trying to log in
 * is in the revoked list. If the user is on the revoked list, an error is
 * sent back to the browser.
 *
 * This filter reads the revoked list file name from the FilterConfig
 * passed in the init() method. It reads the revoked user list file and
 * creates a revokedUsers list.
 *
 * When the doFilter method is called, the user logging in is checked
 * to make sure that the user is not on the revoked Users list.
 *
 */

import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.io.*;

public class LoginFilter implements Filter {

   protected FilterConfig filterConfig;

   java.util.List revokeList;


   /**
    * init() : init() method called when the filter is instantiated.
    * This filter is instantiated the first time j_security_check is
    * invoked for the application (When a protected servlet in the
    * application is accessed).
    */
   public void init(FilterConfig filterConfig) throws ServletException {
      this.filterConfig = filterConfig;

      // read revoked user list
      revokeList = new java.util.ArrayList();
      readConfig();
   }


   /**
    * destroy() : destroy() method called when the filter is taken
    * out of service.
    */
   public void destroy() {
      this.filterConfig = null;
      revokeList = null;
   }

   /**
    * doFilter() : doFilter() method called before the servlet to
    * which this filter is mapped is invoked. Since this filter is
    * mapped to j_security_check,this method is called before
    * j_security_check action is posted.
    */
   public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws java.io.IOException, ServletException {


      HttpServletRequest req = (HttpServletRequest)request;
      HttpServletResponse res = (HttpServletResponse)response;

      // pre login action

      // get username
      String username = req.getParameter("j_username");

      // if user is in revoked list send error
      if ( revokeList.contains(username) ) {
      res.sendError(javax.servlet.http.HttpServletResponse.SC_UNAUTHORIZED);
      return;
      }

      // call next filter in the chain : let j_security_check authenticate
      // user
      chain.doFilter(request, response);

      // post login action

   }

   /**
    * readConfig() : Reads revoked user list file and creates a revoked
    * user list.
    */
   private void readConfig() {
      if ( filterConfig != null ) {

         // get the revoked user list file and open it.
         BufferedReader in;
         try {
               String filename = filterConfig.getInitParameter("RevokedUsers");
               in = new BufferedReader( new FileReader(filename));
         } catch ( FileNotFoundException fnfe) {
               return;
         }

         // read all the revoked users and add to revokeList.
         String userName;
         try {
               while ( (userName = in.readLine()) != null )
                   revokeList.add(userName);
         } catch (IOException ioe) {
         }

      }
```

```
   }

}
```

Important: In the previous code sample, the line that begins `public void doFilter(ServletRequest request` is broken into two lines for illustrative purposes only. The `public void doFilter(ServletRequest request` line and the line after it are one continuous line.

An example of the `web.xml` file that shows the LoginFilter filter configured and mapped to the j_security_check URL:

```
<filter id="Filter_1">
    <filter-name>LoginFilter</filter-name>
    <filter-class>LoginFilter</filter-class>
         <description>Performs pre-login and post-login operation</description>
    <init-param>
    <param-name>RevokedUsers</param-name>
    <param-value>c:\WebSphere\AppServer\installedApps\
                          <app-name>\revokedUsers.lst</param-value>
    </init-param>
</filter-id>

<filter-mapping>
 <filter-name>LoginFilter</filter-name>
     <url-pattern>/j_security_check</url-pattern>
</filter-mapping>
```

An example of a revoked user list file:

```
user1
cn=user1,o=ibm,c=us
user99
cn=user99,o=ibm,c=us
```

Configuring servlet filters for form login processing:

IBM Rational Application Developer or an assembly tool can configure the servlet filters. Two steps are involved in configuring a servlet filter.

Procedure

1. Name the servlet filter and assign the corresponding implementation class to the servlet filter.

   Optionally, assign initialization parameters that get passed to the init method of the servlet filter.After configuring the servlet filter, the `web.xml` application deployment descriptor contains a servlet filter configuration similar to the following example:

```
<filter id="Filter_1">
   <filter-name>LoginFilter</filter-name>
   <filter-class>LoginFilter</filter-class>
   <description>Performs pre-login and post-login
        operation</description>
   <init-param>// optional
     <param-name>ParameterName</param-name>
     <param-value>ParameterName</param-value>
   </init-param>
</filter>
```

2. Map the servlet filter to a URL or a servlet.

   After mapping the servlet filter to a URL or a servlet, the `web.xml` application deployment descriptor contains servlet mapping similar to the following example:

```
<filter-mapping>
   <filter-name>LoginFilter</filter-name>
   <url-pattern>/j_security_check</url-pattern>
               // can be servlet <servlet>servletName</servlet>
</filter-mapping>
```

Example

You can use servlet filters to replace the CustomLoginServlet servlet, and to perform additional authentication, auditing, and logging.

The WebSphere Application Server Samples provide a form login sample that demonstrates how to use the WebSphere Application Server login facilities to implement and configure form login procedures. The sample integrates the following technologies to demonstrate the WebSphere Application Server and Java Platform, Enterprise Edition (Java EE) login functionality:

- Java EE form-based login
- Java EE servlet filter with login
- IBM extension: form-based login

The form login sample is part of the Technology Samples package.

Note: If you install the application server on a z/OS system in which program control is enabled, when you log into a form-based web application you might receive the following error message in the system log file:

```
ICH420I PROGRAM BBORSMCT FROM LIBRARY WAS.SBBOLD2 CAUSED THE ENVIRONMENT TO BECOME UNCONTROLLED.
BPXP014I ENVIRONMENT MUST BE CONTROLLED FOR DAEMON (BPX.DAEMON)
PROCESSING.
```

Although program control is enabled on the z/OS system, the program control extended control bits for the application server's native modules will not be enabled. To prevent or resolve this problem enable the program control bits for all of the native load modules in the SMP/E HTTP Server file system (HFS).

1. Add the necessary attributes to the modules:

```
cd SMPE_ROOT/usr/lpp/install_root/V7R0/lib/modules
extattr +p *
```

2. Add the attributes to the *.so files and bbo* files in the lib directory:

```
cd SMPE_ROOT/usr/lpp/install_root/V7R0/lib
extattr +p *.so
extattr +p bbo*
```

Secure transports with JSSE and JCE programming interfaces

This topic provides detailed information about transport security using Java Secure Socket Extension (JSSE) and Java Cryptography Extension (JCE) programming interfaces. Within this topic, there is a description of the IBM version of the Java Cryptography Extension Federal Information Processing Standard (IBMJCEFIPS).

Java Secure Socket Extension

Java Secure Socket Extension (JSSE) provides the transport security for WebSphere Application Server. JSSE provides the application programming interface (API) framework and the implementation of the APIs for Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, including functionality for data encryption, message integrity, and authentication.

JSSE APIs are integrated into the Java 2 SDK, Standard Edition (J2SDK), Version 5. The API package for JSSE APIs is `javax.net.ssl.*`. Documentation for using JSSE APIs can be found in the J2SE 6 API documentation that is located at `http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`.

Several JSSE providers ship with the Java 2 SDK Version 5 that comes with WebSphere Application Server. The IBMJSSE provider is used in previous WebSphere Application Server releases. Associated with the IBMJSSE provider is the IBMJSSEFIPS provider, which is used when FIPS is enabled on the server. Both of these providers do not work with the Java Message Service (JMS) and HTTP transports in WebSphere Application Server Version 8.5. These transports take advantage of the J2SDK Verison 5 network input/output (NIO) asynchronous channels.

For more information on the new IBMJSSE2 provider, please review the documentation located at `http://www.ibm.com/developerworks/java/jdk/security/60/`.

Customizing Java Secure Socket Extension

You can customize a number of aspects of JSSE by plugging in different implementations of Cryptography Package Provider, X509Certificate and HTTPS protocols, or specifying different default keystore files, key manager factories, and trust manager factories. The following table summarizes which aspects can be customized, what the defaults are, and which mechanisms are used to provide customization.

Table 101. Customizable items. You can customize the following key aspects:

Customizable item	Default	How to customize
X509Certificate	X509Certificate implementation from IBM	The cert.provider.x509v1 security property
HTTPS protocol	Implementation from IBM	The java.protocol.handler.pkgs system property
Cryptography Package Provider	IBMJSSE2	A security.provider.n= line in security properties file. See description.
Default keystore	None	The * javax.net.ssl.keyStore system property
Default truststore	jssecacerts, if it exists. Otherwise, cacerts	The * javax.net.ssl.trustStore system property
Default key manager factory	IbmX509	The ssl.KeyManagerFactory.algorithm security property
Default trust manager factory	IbmX509	The ssl.TrustManagerFactory.algorithm security property

For aspects that you can customize by setting a system property, statically set the system property by using the `-D` option of the Java command. You can set the system property using the administrative console, or set the system property dynamically by calling the java.lang.System.setProperty method in your code: System.setProperty(propertyName,"propertyValue").

For aspects that you can customize by setting a Java security property, statically specify a security property value in the `java.security` properties file. The security property is propertyName=propertyValue. Dynamically set the Java security property by calling the java.security.Security.setProperty method in your code.

The `java.security` properties file is located in the following directory:

app_server_root/`properties` directory.

Application Programming Interface

The JSSE provides a standard application programming interface (API) that is available in packages of the `javax.net` file, `javax.net.ssl` file, and the `javax.security.cert` file. The APIs cover:

- Sockets and SSL sockets
- Factories to create the sockets and SSL sockets
- Secure socket context that acts as a factory for secure socket factories
- Key and trust manager interfaces
- Secure HTTP URL connection classes
- Public key certificate API

Samples using Java Secure Socket Extension

The Java Secure Socket Extension (JSSE) also provides samples to demonstrate its functionality. The Java Secure Socket Extension (JSSE) also provides samples to demonstrate its functionality. You can access the samples in the following location:

Version 1.6

1. Access the http://www.ibm.com/developerworks/java/jdk/security/ website.
2. Click Java 1.6.
3. Click `jssedocs_samples.zip` in the Java Secure Socket Extension (JSSE) Guide section.

Table 102. Extracted files. This table lists the following extracted files:

Files	Description
ClientJsse.java	Demonstrates a simple client and server interaction using JSSE. All enabled cipher suites are used.
OldServerJsse.java	Back-level samples
ServerPKCS12Jsse.java	Demonstrates a simple client and server interaction using JSSE with the PKCS12 keystore file. All enabled cipher suites are used.
ClientPKCS12Jsse.java	Demonstrates a simple client and server interaction using JSSE with the PKCS12 keystore file. All enabled cipher suites are used.
UseHttps.java	Demonstrates accessing an SSL or non-SSL web server using the Java protocol handler of the com.ibm.net.ssl.www.protocol class. The URL is specified with the `http` or `https` prefix. The HTML that is returned from this site is displayed.

See more instructions in the source code. Follow these instructions before you run the samples.

Permissions for Java 2 security

You might need the following permissions to run an application with JSSE: This list is for reference only.
- java.util.PropertyPermission "java.protocol.handler.pkgs", "write"
- java.lang.RuntimePermission "writeFileDescriptor"
- java.lang.RuntimePermission "readFileDescriptor"
- java.lang.RuntimePermission "accessClassInPackage.sun.security.x509"
- java.io.FilePermission "${user.install.root}${/}etc${/}.keystore", "read"
- java.io.FilePermission "${user.install.root}${/}etc${/}.truststore", "read"

For the IBMJSSE provider:
- java.security.SecurityPermission "putProviderProperty.IBMJSSE"
- java.security.SecurityPermission "insertProvider.IBMJSSE"

For the SUNJSSE provider:
- java.security.SecurityPermission "putProviderProperty.SunJSSE"
- java.security.SecurityPermission "insertProvider.SunJSSE"

Debugging

By configuring through the javax.net.debug system property, JSSE provides the following dynamic debug tracing: `-Djavax.net.debug=true`.

A value of `true` turns on the trace facility. Use the administrative console to set the system property for debugging the application server.

Documentation

See the Security: Resources for learning topic for documentation references to JSSE.

JCE

Java Cryptography Extension (JCE) provides cryptographic, key and hash algorithms for WebSphere Application Server. JCE provides a framework and implementations for encryption, key generation, key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block and stream ciphers.

IBMJCE

The IBM version of the Java Cryptography Extension (IBMJCE) is an implementation of the JCE cryptographic service provider that is used in WebSphere Application Server. The IBMJCE is similar to SunJCE, except that the IBMJCE offers more algorithms:
- Cipher algorithm (AES, DES, TripleDES, PBEs, Blowfish, and so on)

- Signature algorithm (SHA1withRSA, MD5withRSA, SHA1withDSA)
- Message digest algorithm (MD5, MD2, SHA1, SHA-256, SHA-384, SHA-512)
- Message authentication code (HmacSHA1, HmacMD5)
- Key agreement algorithm (DiffieHellman)
- Random number generation algorithm (IBMSecureRandom, SHA1PRNG)
- Key store (JKS, JCEKS, PKCS12, JCERACFKS [z/OS only])

The IBMJCE belongs to the com.ibm.crypto.provider.* packages.

For further information, see the information on JCE on the following website: http://www.ibm.com/developerworks/java/jdk/security/60/.

IBMJCEFIPS

The IBM version of the Java Cryptography Extension Federal Information Processing Standard (IBMJCEFIPS) is an implementation of the JCE cryptographic service provider that is used in WebSphere Application Server. The IBMJCEFIPS service provider implements the following:

- Signature algorithms (SHA1withDSA, SHA1withRSA)
- Cipher algorithms (AES, TripleDES, RSA)
- Key agreement algorithm (DiffieHellman)
- Key (pair) generator (DSA, AES, TripleDES, HmacSHA1, RSA, DiffieHellman)
- Message authentication code (MAC) (HmacSHA1)
- Message digest (MD5, SHA-1, SHA-256, SHA-384, SHA-512)
- Algorithm parameter generator (DiffieHellman, DSA)
- Algorithm parameter (AES, DiffieHellman, DES, TripleDES, DSA)
- Key factory (DiffieHellman, DSA, RSA)
- Secret key factory (AES, TripleDES)
- Certificate (X.509)
- Secure random (IBMSecureRandom)

Application Programming Interface

Java Cryptography Extension (JCE) has a provider-based architecture. Providers can be plugged into the JCE framework by implementing the APIs that are defined by the JCE. The JCE APIs cover:

- Symmetric bulk encryption, such as DES, RC2, and IDEA
- Symmetric stream encryption, such as RC4
- Asymmetric encryption, such as RSA
- Password-based encryption (PBE)
- Key agreement
- Message authentication codes

Samples using Java Cryptography Extension

There are samples located on the `http://www.ibm.com/developerworks/java/jdk/security/` website in the `jceDocs_samples.zip` file. Unzip the file and locate the following samples in the `jceDocs/samples` directory:

Table 103. Samples using Java Cryptography Extension. This table describes samples using Java Cryptography Extension.

File	Description
SampleDSASignature.java	Demonstrates how to generate a pair of DSA keys (a public key and a private key) and use the key to digitally sign a message using the SHA1withDSA algorithm
SampleMarsCrypto.java	Demonstrates how to generate a Mars secret key, and how to do Mars encryption and decryption

Table 103. Samples using Java Cryptography Extension (continued). This table describes samples using Java Cryptography Extension.

File	Description
SampleMessageDigests.java	Demonstrates how to use the message digest for MD2 and MD5 algorithms
SampleRSACrypto.java	Demonstrates how to generate an RSA key pair, and how to do RSA encryption and decryption
SampleRSASignatures.java	Demonstrates how to generate a pair of RSA keys (a public key and a private key) and use the key to digitally sign a message using the SHA1withRSA algorithm
SampleX509Verification.java	Demonstrates how to verify X509 certificates

Documentation

Refer to the Security: Resources for learning topic for documentation on JCE.

Using System Authorization Facility keyrings with Java Secure Sockets Extension

WebSphere Application Server for z/OS customers running server W50100*x* or later, with Java Development Kit 1.3 level SR20 or later, can modify their WebSphere Application Server systems to use System Authorization Facility (SAF) for Java Secure Sockets Extension (JSSE) as well as Secure Sockets Layer (SSL), which eliminates the need to maintain duplicate certificates in the hierarchical file system (HFS).

Before you begin

WebSphere Application Server for z/OS running at maintenance levels before W502000 stored digital certificate information in two different places because of the following Software Development Kit (SDK) restrictions:

- JSSE used digital certificates stored in hierarchical file system files
- SSL used digital certificate information stored in the SAF database

Systems customized at W502000 or above use the single SAF digital certificate repository by default, and do not need the modifications described below.

About this task

WebSphere Application Server for z/OS customers running server W50100x or later, with Java Development Kit 1.3 level SR20 or later, can modify their WebSphere Application Server systems to use SAF for JSSE as well as SSL (eliminating the need to maintain duplicate certificates in the HFS). The instructions below describe how to enable this support.

Note: Systems that are customized at maintenance levels at or after W502000 use the single (SAF digital certificate repository by default, and these systems do not need the modifications described below.

To use SAF certificates with JSSE:

Procedure

1. Update the Java Management Extensions (JMX) connector settings to indicate the SAF keyring names for the node.
   a. Log in to the administrative console using an identity with administrator authority.
   b. Click **Servers > Application servers >** *server_name*.
   c. Under Server infrastructure, click **Administration > Administration services**.
   d. Under Additional properties, click **JMX connectors**.
   e. On the JMX Connectors panel, click **SOAPConnector**.

f. Under Additional Properties, click **Custom Properties**.

g. On the Custom properties page, click **sslConfig**.

h. On the sslConfig page, look at the Value field. Verify that this field says *node_name/DefaultSSLSettings*, where *nodename* represents the node name where the application server resides. Record the node name for a subsequent step.

i. Select ***node_name*/RACFJSSESettings** from the list next to the Value field, where *node_name* is the same as the node name that you previously recorded.

j. Click **OK**. The Custom Properties page appears with a message indicating that changes are made to your local configuration. Do not click **Save** because additional changes that are required.

2. Click **Servers > Application servers** and repeat the previous substeps for each of the other application servers in the cell.

3. Update the Java Management Extensions (JMX) connector settings to indicate the SAF keyring names for the deployment manager node.

a. Click **System administration > Deployment manager**.

b. Under Additional properties, click **Administration services > JMX Connectors**.

c. On the JMX Connectors panel, click **SOAPConnector**.

d. Under Additional properties, click **Custom properties**.

e. On the Custom properties page, click **sslConfig**.

f. On the sslConfig page, look at the Value field. This field displays *dmnode/DefaultSSLSettings*, where *dmnode* represents the deployment manager node name. Record the node name for a subsequent step.

g. Select **dmnode/RACFJSSESettings** from the list next to the Value field, where **dmnode** represents the Deployment Manager node name.

h. Click **OK**. After a short time the Custom Properties page appears with a message at the top indicating that changes have been made to your local configuration. Do not click **Save** at this point because there are additional changes that are required.

4. Update the Java Management Extensions (JMX) connector settings to indicate the SAF keyring names for the node agent.

a. Click **System administration > Node agents >** *Node_name*. Record the node agent name for the next step.

b. Under Additional properties, click **Administration services > JMX Connectors**.

c. On the JMX Connectors panel, click **SOAPConnector**.

d. Under Additional properties, click **Custom properties**.

e. On the Custom properties page, click **sslConfig**.

f. On the sslConfig page, look at the Value field. This field displays *nodename/DefaultSSLSettings*, where *nodename* is the node name where the node agent resides. Record the node name for a subsequent step.

g. Select **nodename/RACFJSSESettings** from the list next to the Value field, where **nodename** is the node name that you previously recorded.

h. Click **OK**. The Custom Properties page is displayed with a message indicating that changes have been made to the local configuration. Do not click **Save** at this point because additional changes are required.

5. Click **System administration > Node agents** and repeat the previous substeps for each of the other node agents servers in the cell.

6. Click **Save** when the `Changes have been made to your local configuration. Click Save to apply changes to the master configuration` message is displayed.

7. On the Save page, select the **Synchronize changes with Nodes** option and click **Save**. After the changes are saved, the administrative console returns to the home page.

8. Update the soap.client.props file in the *profile_root*/properties directory to indicate the SAF keyring names that are appropriate for your configuration. The soap.client.props file is used by the wsadmin.sh script and is located in the application server or deployment manager (user.install.root)/properties file. The purpose of the soap.client.props file is to specify the values used by SOAP clients such as wsadmin.sh. In a cell configured before WebSphere Application Server for z/OS maintenance level W502000, the soap.client.props file indicates the names of the Java key stores used by JSSE. Once your cell is using SAF keyrings for JSSE administration, verify that SAF keyrings are being used for SOAP clients.

   The soap.client.props file is used by the wsadmin.sh script.

   Changes to wsadmin client SAF keyrings require updates to the soap.client.props file and the creation of a keyring for administrators. Specify the following values:

```
com.ibm.ssl.protocol=SSL
com.ibm.ssl.keyStoreType=JCERACFKS
com.ibm.ssl.keyStore=safkeyring:///yourkeyringName
com.ibm.ssl.keyStorePassword=password
com.ibm.ssl.trustStoreType=JCERACFKS
com.ibm.ssl.trustStore=safkeyring:///yourKeyringName
com.ibm.ssl.trustStorePassword=password
```

   =

   The password value specified does not represent a real password because you can use any string. Replace the string yourKeyringName with your administrative SAF keyring. The keyring name used by all WebSphere administrators and the administrative started task user ID (default WSADMSH) must be the same. Additionally, a keyring must be created for each user that uses the wsadmin.sh file with the SOAP connector when using SAF keyrings and security is enabled. (A keyring is created by the customization process for your initial administrative user ID, such as WSADMIN.)

   A description of how to create keyrings for administrative users in SAF is described in SSL considerations for WebSphere Application Server administrators.

9. Recycle the cell.

Configuring Federal Information Processing Standard Java Secure Socket Extension files

Use this topic to configure Federal Information Processing Standard Java Secure Socket Extension files.

About this task

In WebSphere Application Server, the Java Secure Socket Extension (JSSE) provider used is the IBMJSSE2 provider. This provider delegates encryption and signature functions to the Java Cryptography Extension (JCE) provider. Consequently, IBMJSSE2 does not need to be Federal Information Processing Standard (FIPS)-approved because it does not perform cryptography. However, the JCE provider requires FIPS-approval.

WebSphere Application Server provides a FIPS-approved IBMJCEFIPS provider that IBMJSSE2 can utilize. The IBMJCEFIPS provider that is shipped in WebSphere Application Server Version 8.5 supports the following SSL ciphers:

- SSL_RSA_WITH_AES_128_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_AES_128_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_AES_128_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA

Even though the IBMJSSEFIPS provider is still present, the runtime does not use this provider. If IBMJSSEFIPS is specified as a contextProvider, WebSphere Application Server automatically defaults to the IBMJSSE2 provider (with the IBMJCEFIPS provider) for supporting FIPS. When enabling the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option on the server SSL certificate and key management panel, the runtime always uses IBMJSSE2, despite the contextProvider that you specify for SSL (IBMJSSE, IBMJSSE2 or IBMJSSEFIPS). Also, because FIPS requires the SSL protocol be TLS, the runtime always uses TLS when FIPS is enabled, regardless of the SSL protocol setting in the SSL repertoire. This simplifies the FIPS configuration in Version 8.5 because an administrator needs to enable only the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option on the server SSL certificate and key management panel to enable all transports using SSL.

Procedure

1. Click **Security > SSL certificate and key management > Manage FIPS**.
2. Select the **Enable FIPS 140-2** option and click **Apply**. This option makes IBMJSSE2 and IBMJCEFIPS the active providers.
3. Accommodate Java clients that must access enterprise beans.

   Change the `com.ibm.security.useFIPS` property value from `false` to `true` in the *profile_root*`/properties/ssl.client.props` file.
4. Ensure that the com.ibm.ssl.protocol property within the *profile_root*`/properties/ssl.client.props` file is set to TLS.
5. Ensure that the `java.security` file includes the provider.

What to do next

After completing these steps, a FIPS-approved JSSE or JCE provider offers increased encryption capabilities. However, when you use FIPS-approved providers:

- By default, Microsoft Internet Explorer might not have TLS enabled. To enable TLS, open the Internet Explorer browser and click **Tools > Internet Options**. On the Advanced tab, select the Use TLS 1.0 option.

  Note: Netscape Version 4.7.*x* and earlier versions might not support TLS.
- If you have an administrative client that uses a SOAP connector and you enable FIPS, add the following line to the *profile_root*`/properties/soap.client.props` file:

  `com.ibm.ssl.contextProvider=IBMJSSEFIPS`
- When you select the **Use the Federal Information Processing Standard (FIPS)** option on the SSL certificate and key management panel, the Lightweight Third-Party Authentication (LTPA) token format is not backwards-compatible with previous releases of WebSphere Application Server. However, you can import the LTPA keys from a previous version of the application server.

Note: When enabling FIPS, you cannot configure cryptographic token devices in the SSL repertoires. IBMJSSE2 must use IBMJCEFIPS when utilizing cryptographic services for FIPS.

The following FIPS 140-2 approved cryptographic providers that are the only devices that are supported with the FIPS option:

- IBMJCEFIPS (certificate 376)
- IBM Cryptography for C (ICC) (certificate 384)

The relevant certificates are listed on the NIST website: Cryptographic Module Validation Program FIPS 140-1 and FIPS 140-2 Pre-validation List

To unconfigure the FIPS provider, reverse the changes that you made in the previous steps. After you reverse the changes, verify that you have made the following changes to the `sas.client.props`, `soap.client.props`, and `java.security` files:

- In the `ssl.client.props` file, you must change the `com.ibm.security.useFIPS` value to `false`.
- In the `java.security` file, you must change the FIPS provider to a non-FIPS provider.

  If you are using the IBM SDK `java.security` file, you must change the first provider to a non-FIPS provider as shown in the following example:

```
#security.provider.1=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.1=com.ibm.crypto.provider.IBMJCE
security.provider.2=com.ibm.jsse.IBMJSSEProvider
security.provider.3=com.ibm.jsse2.IBMJSSEProvider2
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
#security.provider.6=com.ibm.crypto.pkcs11.provider.IBMPKCS11
```

  If you are using the Sun JDK `java.security` file, you must change the third provider to a non-FIPS provider as shown in the following example:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.ibm.security.jgss.IBMJGSSProvider
security.provider.3=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.4=com.ibm.crypto.provider.IBMJCE
security.provider.5=com.ibm.jsse.IBMJSSEProvider
security.provider.6=com.ibm.jsse2.IBMJSSEProvider2
security.provider.7=com.ibm.security.cert.IBMCertPath
#security.provider.8=com.ibm.crypto.pkcs11.provider.IBMPKCS11
```

When you use the FIPS provider, the IBM Software Development Kit (SDK) might issue an error message that refers to a bad certificate. Although this error message can result from a multitude of reasons, review your security configuration and consider one of the following actions:

- Reduce the cipher suite level to Medium, if your cipher suite level is currently Strong.

  Note: You can change the cipher suite level for different levels of your environment such as the node or server level. Limit the change to the level of your environment where the change is necessary.

  To change the cipher suite, see the cipher suite groups information within the quality of protection settings documentation. If you change the cipher suite level to Medium, save and synchronize the changes. If the **Dynamically update the run time when SSL configuration changes occur** option is selected, you do not need to restart the server. However, if the option is not selected, you must restart the server for the changes to be effective. The **Dynamically update the run time when SSL configuration changes occur** option is available within the administrative console on the SSL certificate and key management panel. To access the panel, click **Security** > **SSL certificate and key management**.
- Install security level 3 FMID JCPT3A1 for the z/OS operating system.

  Security Level 3 FMID JCPT3A1 is the z/OS operating system implementation of the FIPS 140-2 approved cryptographic providers.

WebSphere Application Server security standards configurations

WebSphere Application Server can be configured to work with various security standards, which are typically used to meet security requirements required by the government.

Note: WebSphere Application Server integrates cryptographic modules, which include Java Secure Socket Extension (JSSE) and Java Cryptography Extension (JCE). Most of the requirements in the standards are handled in the JSSE and JCE, which must undergo the certification process to meet government standards. WebSphere Application Server must be configured to run with the JSSE and JCE enabled for a particular standard, and now supports the FIPS 140-2, SP800-131 and Suite B security standards.

- **FIPS 140-2** are Federal Information Processing Standards (FIPS) that specify requirements on cryptographic modules. WebSphere Application Server has been able to configure using this standard the longest. Many users can be configured to use this level, but might be required to move up to the newer SP800-131 or Suite B standard.

  See The National Institute of Standards and Technology web site for more information about the 140-2 standard.

To configure FIPS 140-2, see the topic “Configuring Federal Information Processing Standard Java Secure Socket Extension files”.

- **SP800-131** is a requirement originated by the National Institute of Standards and Technology (NIST) which requires longer key lengths and stronger cryptography. The specification also provides a transition configuration to enable users to move to a strict enforcement of SP800-131. The transition configuration also enables users to run with a mixture of settings from both FIPS140-2 and SP800-131. SP800-131 can be run in two modes, `transition` and `strict`.

  Strict enforcement of SP800-131 requirements on WebSphere Application server includes the following:

  – The use of the TLSv1.2 protocol for the Secure Sockets Layer (SSL) context.
  – Certificates must have a minimum length of 2048. Elliptical Curve (EC) certificate require a minimum size of 244-bit curves.
  – Certificates must be signed with a signature algorithm of SHA256, SHA384, or SHA512. Valid signatureAlgorithms include:
    - SHA256withRSA
    - SHA384withRSA
    - SHA512withRSA
    - SHA256withECDSA
    - SHA384withECDSA
    - SHA512withECDSA
  – SP800-131 approved Cipher suites

  See The National Institute of Standards and Technology web site for more details about the SP800-131 standard.

  See the topic “Transitioning WebSphere Application Server to the SP800-131 security standard” for information on how to transition WebSphere Application Server to the SP800-131 strict standard. See the topic “Configuring WebSphere Application Server for SP800-131 standard strict mode” for information on how to configure SP800-131.

- **Suite B** is a requirement originated by the National Security Agency (NSA) to specify a cryptographic interoperability strategy. This standard is similar to SP800-131 with some tighter restrictions. Suite B can run in 2 modes: 128-bit or 192-bit. If using 192-bit mode, users must apply the unrestricted policy file to the JDK so that the stronger cipher required for the 192-bit mode can be used.

  See the topic "Configuring WebSphere Application Server for the Suite B security standard" for information on to configure Suite B.

  Suite B requirements on WebSphere Application Server includes the following:

  – The use of the TLSv1.2 protocol for the SSL Context
  – Suite B approved Cipher suites
  – Certificates:
    - 128 bit mode certificates must be signed with SHA256withECDSA
    - 192 bit mode certificates must be signed with SHA384withECDSA
  – Ciphers:
    - SSL_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
    - SSL_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384.

Properties used to enable the Security Standards

The IBM virtual machine for Java (JVM) runs in a given security mode based on system properties. WebSphere Application Server sets these system properties based on security configuration settings. The security configuration can be set up through the administrative console or through scripting admin tasks. If an application sets these properties directly it can affect WebSphere Application Server SSL communication.

Table 104. JVM system properties to enable the security standard

Security standard	System property to enable	Valid values
FIPS 140-2	com.ibm.jsse2.usefipsprovider	`true` or `false`
SP800-131	com.ibm.jsse2.sp800-131	`transition` or `strict`
Suite B	com.ibm.jsse2.suiteB	128 or 192

WebSphere Application Server configuration clears out all of these properties if they are set, then sets them to how the security configuration is specified. WebSphere Application Server enables the security standard based on the custom properties set in the security configuration.

WebSphere Application Server security custom properties to enable the security standard

Table 105. WebSphere Application Server security custom properties to enable the security standard

Security standard	Security custom properties	JVM system property
FIPS 140-2	com.ibm.security.useFips=true com.ibm.websphere.security.FIPSLevel=FIPS140-2	com.ibm.jsse2.usefipsprovider=true
SP800-131- transition	com.ibm.security.useFips=true com.ibm.websphere.security.FIPSLevel=transition	com.ibm.jsse2.sp800-131=transition
SP800-131 – strict	com.ibm.security.useFips=true com.ibm.websphere.security.FIPSLevel=SP800-131	com.ibm.jsse2.sp800-131=strict
Suite B 128	com.ibm.security.useFips=true com.ibm.websphere.security.suiteB=128	com.ibm.jsse2.suiteB=128
Suite B 192	com.ibm.security.useFips=true com.ibm.websphere.security.suiteB=192	com.ibm.jsse2.suiteB=192

Convert certificates

Use this page to convert certificates to the selected security standard. All certificates in keystores associated with an Secure Socket Layer (SSL) configuration are converted.

To view this administrative console page, click **Security > SSL certificate and key management > Manage FIPS > Convert certificates**.

Algorithm: Specifies the signature algorithm used to convert the certificate to the selected security standard.

The following choices are available:

Strict Select for the strict enforcement of the SP800-131 standard.

Strict enforcement of SP800-131 requirements on WebSphere Application Server includes the following:

- The use of the TLSv1.2 protocol for the Secure Sockets Layer (SSL) context.
- Certificates must have a minimum length of 2048. Elliptical Curve (EC) certificate require a minimum size of 244-bit curves.
- Certificates must be signed with a signature algorithm of SHA256, SHA384, or SHA512. Valid signatureAlgorithms include:
  - SHA256withRSA
  - SHA384withRSA
  - SHA512withRSA
  - SHA256withECDSA
  - SHA384withECDSA
  - SHA512withECDSA

- SP800-131 approved Cipher suites

Suite B with 128 bit keys
This requirement places some tighter restrictions on the SP800-131 specification. 128-bit mode certificates must be signed with SHA256withECDSA.

Suite B with 192 bit keys
192 bit mode certificates must be signed with SHA384withECDSA.

To run in 192-bit mode, the unrestricted policy files must be in place on the JDK.

New certificate key size: Specifies the key size to use when converting the certificates.

The valid values are 512, 1024, 2048, 4096 and 8192. The default value is 2048.

Note: Elliptical Curve signature algorithms require specific sizes, so you must provide a size.

Certificates that can not be converted: Lists the certificates that are not compliant with the specified security standard and cannot be converted.

If certificates show up listed in this box, the server is unable to convert the certificates for you. You must replace these certificates with ones that meet Suite B requirements. Reasons why the server cannot convert the certificates might include:

- The certificate was created by a Certificate Authority (CA).
- The certificate is in a read-only keystore.

Manage FIPS

Use this page to disable Federal Information Processing Standards (FIPS) or to enable security standards that are required by the government.

WebSphere Application Server integrates cryptographic modules, which include Java Secure Socket Extension (JSSE) and Java Cryptography Extension (JCE). JSSE and JCE must undergo the certification process to meet government standards, and WebSphere Application Server must be configured to use them as specified by the standards.

To view this administrative console page, click **Security > SSL certificate and key management > Manage FIPS**.

Disable FIPS: Select to disable FIPS, which is the default.

Data type:	Default:	Range:
Boolean	Enabled	Enabled or Disabled

Enable FIPS 140-2: Select to enable FIPS 140-2. This option makes IBMJSSE2 and IBMJCEFIPS the active providers.

Federal Information Processing Standards (FIPS) specifies requirements on cryptographic modules. WebSphere Application Server has been able to configure using the FIPS 140-2 standard the longest. Many users can be configured to use this level, but might be required to move up to the newer SP800-131 or Suite B standard.

Data type:	Default:	Range:
Boolean	Enabled	Enabled or Disabled

Enable SP800-131: Select to enable SP800-131.

SP800-131 is a requirement originated by the National Institute of Standards and Technology (NIST) which requires longer key lengths and stronger cryptography. The specification also provides a transition configuration to enable users to move to a strict enforcement of SP800-131. The transition configuration also enables users to run with a mixture of settings from both FIPS140-2 and SP800-131. SP800-131 can be run in two modes, `transition` and `strict`.

Data type:	Default:	Range:
Boolean	Enabled	Enabled or Disabled

Enable Suite B: Accept 128 bit keys: Select to specify that suite B cryptography is used, and is configured to accept a 128-bit key size. Keystore certificate algorithms require Elliptical curve (EC) cryptography.

Data type:	Default:	Range:
Boolean	Enabled	Enabled or Disabled

Enable Suite B: Accept 192-bit keys: Select to specify that suite B cryptography is used, and is configured to accept a 192-bit key size. Keystore certificate algorithms require Elliptical curve (EC) cryptography.

Suite B can run in 2 modes: 128-bit or 192-bit. If using 192-bit mode, you must apply the unrestricted policy file to the JDK so that the stronger cipher required for the 192-bit mode can be used.

Data type:	Default:	Range:
Boolean	Enabled	Enabled or Disabled

Convert certificates: Select to convert certificates to the selected security standard. All certificates in keystores associated with an Secure Socket Layer (SSL) configuration are converted.

Configuring WebSphere Application Server for the Suite B security standard

You can configure WebSphere Application Server to use the new Suite B security standard.

Before you begin

Read the “WebSphere Application Server security standards configurations” topic for more background information regarding security standards.

About this task

The National Security Agency (NSA) created a cryptographic interoperability strategy called Suite B. It places specific requirements on the National Institute of Standards and Technology (NIST) SP800-131 standard.

Suite B requirements:

WebSphere Application Server must be compliant with the following Suite B requirements:

- SSL configuration must use the TLSv1.2 protocol.
- The com.ibm.jsse.suiteb system property must be set to 128 or 192.
- Certificates running in 128-bit mode must be created with the SHA256withECDSA signature algorithm. Certificates running in 192-bit mode must be created with the SHA384withECDSA signature algorithm.

  Note: To run in 192-bit mode, the unrestricted policy files must be in place on the JDK.

- Suite B approved cipher suites must be used.

To configure the server for the Suite B standard:

Procedure

1. Click **Security > SSL certificate and key management > Manage FIPS** To run in a Suite B mode, all of the certificates used for SSL on the server must be converted to certificates that comply with Suite B requirements.
2. To convert certificates, under Related Items click **Convert Certificates**.
3. Select the radio button labeled **128-bit or 192-bit** in the Algorithm box.

   Note: Elliptical Curve signature algorithms require specific sizes, so you must provide a size.
4. Click **Apply/Save**. If no certificates show up in the box labeled **Certificates that can not be converted**, then you can enable the standard.

   If certificates show up listed in the box labeled **Certificates that can not be converted**, the server is unable to convert the certificates for you. You must replace these certificates with ones that meet Suite B requirements. Reasons why the server cannot convert the certificates might include:
   - The certificate was created by a Certificate Authority (CA).
   - The certificate is in a read-only keystore.

   After certificates are converted to meet the Suite B specifications, follow the remaining steps to enable the Suite B standard.
5. Click **SSL certificate and key management > Manage FIPS**.
6. Select the **Suite B: Accept 128 bit key** for 128-bit mode or the **Suite B: Accept 192 bit key** for 192-bit mode.
7. Click **Apply/Save**.
8. Restart the servers and manually sync the nodes for the Suite B standard to take effect.

   When these changes are applied and the server is restarted, the SSL configurations on the server is modified to use the TLSv1.2 protocol, and the com.ibm.jsse.suiteb system property is set to the desired Suite B mode. The SSL configuration uses the appropriate SSL ciphers for the standard.

   There are wsadmin tasks also available that can enable the Suite B standard using scripting. :
   - Check the status of certificates for the security standard by using the listCertStatusForSecurityStandard task.
   - Convert certificates for the security standard by using the convertCertForSecurityStandard task.
   - Enable the security standard by using the enableFips task.
   - To see the security standard setting, use the getFipsInfo task.
9. Once the server is configured for SP800-131 strict mode, the `ssl.client.props` file must be modified so that administrative clients are running in SP800-131 strict mode. They are unable to make a SSL connection to the server with the change. Edit the `ssl.client.props` file by doing the following:
   a. Modify com.ibm.security.useFIPS to be set to `true`.
   b. Add the com.ibm.jsse.suiteb property, and set it to 128 or 192.
   c. Change the com.ibm.ssl.protocol property to TLSv1.2.

What to do next

The Suite B standard requires that the SSL connection use the TLSv1.2 protocol. For a browser to access the administrative console or an application, the browser must support and first be configured to use the TLSv1.2 protocol.

Note: When enabling the security standards on a Network Deployed, the node and deployment manager can be in an incompatible protocol state. Since configuring the security standard requires the server

to be restarted, it is recommended that all node agents and servers be stopped, leaving the deployment manager running. Once the configuration changes are made through the console, restart the deployment manager.

Manually sync the nodes with syncNode, and start the node agents and servers. To use syncNode, you might need to update the `ssl.client.props` file to communicate with the deployment manager.

Transitioning WebSphere Application Server to the SP800-131 security standard

The National Institute of Standards and Technology (NIST) Special Publications 800-131 standard strengthens algorithms and increases the key lengths to improve security. The standard also provides for a transition period to move to the new standard. You can configure WebSphere Application Server for SP800-131 standard transition mode.

Before you begin

Read the "WebSphere Application Server security standards configurations" topic for more background information regarding security standards.

About this task

The transition period enables a user to run in a mixed environment of settings not supported under the standard along with those that are supported. The NIST SP800-131 standard requires that users be configured for strict enforcement of the standard by a specific timeframe. See The National Institute of Standards and Technology web site for more details.

The transition options can be very useful when trying to get to `strict` SP800-131. The servers can accept a mixture old settings and new requirements. For example. they can convert certificates but continue to use the TLS protocol.

WebSphere Application Server can be configured to run SP800-131 in a `transition` mode or a `strict` mode. For information on how to configure strict mode. read the Configuring WebSphere Application Server for strict mode SP800-131 security standard topic.

To run in the SP800-131 transition mode, the server must be in a specific configuration setting as well. Other strict requirements can be include as wanted.

- The com.ibm.jsse2.sp800-131 system property must be set to `transition` for the JSSE to run in the transition mode.
- The SSL configuration protocols must be one of the TLS settings. Valid values include TLS, TLSv1, TLSv1.1, and TLSv1.2.

Procedure

1. Click **Security > SSL certificate and key management > Manage FIPS**.
2. Select the **Enable SP800-131** radio button.
3. Select the **Transition** radio button.
4. You have the choice to change the protocols in SSL configuration to TLSv1.2 by optionally selecting the **Update the SSL configuration to require TLSv1.2** box. If you do not select this box, all SSL configurations are set to TLS.
5. Click **Apply/Save**.
6. Restart the servers.

When these changes are applied, and the server is restarted, all of the SSL configuration on the server are modified to use the TLS or TLSv1.2 protocol, and the com.ibm.jsse2.sp800-131 system property is set to `transition`. The SSL configuration uses the appropriate SSL ciphers for the standard.

Before you can move to the strict mode certificate, the protocol in the configuration must meet the strict requirements.

You can go to directly to the SSL configuration and set protocols to TLSv1.2 by doing the following:

7. Click **Security > SSL certificate and key management > SSL Configurations**.
8. Select a SSL configuration from the collection panel.
9. Under Related Items, select **Quality of protection (QoP)**.
10. Select **TLSv1.2** from the pull-down box labeled Protocol
11. Click **Apply/Save**. To change the SSL protocol using scripting, the modifySSLConfig task can also be used.

    Certificates must have a minimum size of 2048 (244 if an Elliptical Curve certificate), and signed with SHA256, SHA384, or SHA512. You can create new ones on the console and replace the old one, or import certificates that meet the standards requirements.

    There are a number of options you can use to replace certificates.

    - Use the Convert Certificate panel. This panel converts all certificates to meet the standard specified.
      a. Click **Security > SSL certificate and key management > Manage FIPS > Convert Certificate**

         Note: If there are any certificates in the box labeled **Certificates** that can not be converted, then a certificate can not be converted using this option.
      b. Select the **Strict** radio button and choose which signatureAlgorithm to use when creating the new certificates from the pull-down box.
      c. Select the size of the certificate from the pull-down box labeled **New Certificate Key Size**. Note that Elliptical Curve signature algorithms require a specific size, so there is no need to provide a size.
      d. Click **Apply/Save**.

         The convertCertForSecurityStandard scripting task can also be used to convert all certificates to meet a specified standard.
    - Use the personal certificate panels to create new certificates and replace a certificate that does not meet the requirements by doing the following:
      a. Click **Security > SSL certificate and key management > Key stores and certificates**.
      b. Select a keystore from the collection panel.
      c. Select **Personal Certificate**.
         1) From the pull-down list on the Create button, select **Self-Signed Certificate**.
         2) Fill in an alias for the certificate. Select a signature algorithm for the certificate that is signed with SHA256, SHA384, or SHA512. Choose a size that is 2048 or greater. Note that Elliptical Curve signature algorithms require a specific size, so there is no need to specify a size.
         3) Click **Apply/Save**.
         4) Go back to the Personal certificate collection panel and select the certificate that does not meet the standard. Click **Replace**.
         5) On the Replace panel, select the certificate created that meets the standard from the pull-down list in the box labeled **Replace with**.
         6) Select **Delete old certificate** after replacement, and **Delete old signer boxes**.
         7) Click **Apply/Save**.

Note: To replace chained certificates, a root certificate must be created that meets the standard. Follow the previous navigation path to the root certificate in the defaultRootStore, then create a chained certificate with that new root certificate.

The createSelfSigneCertificate scripting task can also be used to create self-signed certificate. The replaceCertificate scripting task can also be used to replace the new certificate for the old one.

- Use the personal certificate panels to import certificates and to replace the certificate that does not meet the requirements. Some certificate come from external sources such as a Certificate Authority (CA).
  a. Click **Security > SSL certificate and key management > Key stores and certificates**.
  b. Select a keystore from the collection panel.
  c. Select **Personal Certificate**.
     1) Select **Import Certificate**.
     2) Fill in the information needed to access the certificate in an existing keystore file.
     3) Click **Apply/Save**.
     4) Go back to the Personal certificate collection panel and select the certificate that does not meet the standard. Click the **Replace** button.
     5) On the Replace panel, select the certificate created that meets the standard from the pull-down list in the box labeled **Replace with**. Select **Delete old certificate after replacement** and **Delete old signer boxes**.
     6) Click **Apply/Save**.

        The importCertificate scripting task can also be used to import a certificate. The replaceCertificate scripting task can also be used to replace the new certificate for the old one.

12. To enable strict SP800-131, click **Security > SSL certificate and key management > Manage FIPS**.
13. Click the **Enable SP800-131**.
14. Click the **Strict**.
15. Click **Apply/Save**.
16. Restart your servers and manually sync your nodes for the changes to take effect.
17. Configure the client `ssl.client.props` file for the transition mode SP800-131 standard. Once the server is configured for SP800-131 transition mode, the `ssl.client.props` file might need to modified so that the admin client can connect to the server.

    Edit the `ssl.client.props` file. Change the com.ibm.ssl.protocol property to match the protocol the server is using.
18. Configure the `client ssl.client.props` file for strict mode SP800-131 standard. Once the server is configured for SP800-131 strict mode, the `ssl.client.props` file must be modified so that the admin client is running in SP800-131 strict mode. It is not able to make a SSL connection to the server without the change.

    Edit the `ssl.client.props` file as follows:
    a. Modify the com.ibm.security.useFIPS to be set to`true`.
    b. Add the com.ibm.websphere.security.FIPSLevel=SP800-131 just below the useFips property.
    c. Change the com.ibm.ssl.protocol property to TLSv1.2

What to do next

The browser used to access the administrative console or an application must use a protocol that is compatible with the server. If the server is running in a transition mode, the browser must be set to use the protocol that matches the server. The SP800-131 standard requires that the SSL connection use the TLSv1.2 protocol, so the browser must support TLSv1.2 and use it to access the administrative console.

Configuring WebSphere Application Server for SP800-131 standard strict mode

You can configure WebSphere Application Server to use the SP800-131 standard strict mode.

Before you begin

Read the “WebSphere Application Server security standards configurations” topic for more background information regarding security standards.

About this task

The National Institute of Standards and Technology (NIST) Special Publications (SP) 800-131 standard strengthens algorithms and increases the key lengths to improve security. The standard also provides for a transition period to move to the new standard. The transition period enables a user to run in a mixed environment of settings not supported under the standard along with those that are supported. The NIST SP800-131 standard requires that users be configured for strict enforcement of the standard by a specific timeframe. See The National Institute of Standards and Technology web site for more details.

WebSphere Application Server can be configured to run SP800-131 in a `transition` mode or a `strict` mode. For instructions on how to configure transition mode, read the topic “Transitioning WebSphere Application Server to the SP800-131 Security Standard”.

To run in strict mode, there are several changes necessary to the server configuration:

- Secure Sockets Layer (SSL) configuration must use the TLSv1.2 protocol.
- The com.ibm.jsse2.sp800-131 system property must be set to `strict` for the JSSE to run in a strict SP800-131 mode.
- Certificates used for SSL communication must have a minimum length of 2048, and for Elliptical Curve (EC) certificates they must have a minimum length of 244.
- Certificates must be signed with a signature algorithm of SHA256, SHA384, or SHA512.
- SP800-131 approved cipher suites must be used.

.

Procedure

1. Click **Security > SSL certificate and key management > Manage FIPS** To run in a strict SP800-131 mode, all of the certificates used for SSL on the server must be converted to certificates that comply with the SP800-131 requirements.
2. To convert certificates, under Related Items, click **Convert Certificates**.
3. Select the radio button marked **Strict**, and choose which signatureAlgorithm to use when creating the new certificates from the pull-down box.
4. Select the size of the certificate from the pull-down box labeled **New Certificate Key Size**.

   Note: If you choose an Elliptical Curve signature algorithm, they require specific sizes; you are not able to fill in a size. The correct size is used instead.
5. If no certificates are displayed in the box labeled **Certificates that can not be converted**, click **Apply/Save**.
6. If certificates are displayed in the box labeled **Certificates that can not be converted**, the server is unable to convert the certificate for you. You must replace these certificates with ones that meet SP800-131 requirements. Reasons why the server can not convert a certificate for you include:
   - The certificate was created by a Certificate Authority (CA)
   - The certificate is in a read only keystore

Once certificates are converted to meet the SP800-131 specification, perform the following steps to enable SP800-131 strict mode.

7. Click **SSL certificate and key management > Manage FIPS**.
8. Enable the radio button labeled **Enable SP800-131**.
9. Enable the radio button labeled **Strict**.
10. Click **Apply/Save**.
11. Restart the servers and manually sync the nodes for the SP800-131 strict mode to take effect.

    When these changes are applied, and the server is restarted, all of the SSL configuration on the server are modified to use the TLSv1.2 protocol and the com.ibm.jsse2.sp800-131 system property is set to `strict`. The SSL configuration uses the appropriate SSL ciphers for the standard.

    There are several wsadmin tasks that can be used to enable strict SP800-131 using scripting

    - Check the status of certificates for the security standard by using the listCertStatusForSecurityStandard task.
    - Convert certificates for the security standard by using the convertCertForSecurityStandard task.
    - Enable the security standard by using the enableFips task.
    - To see the security standard setting, use the getFipsInfo task.
12. Once the server is configured for SP800-131 strict mode, the `ssl.client.props` file must be modified so that the admininstrative client is running in SP800-131 strict mode. They are not able to make a SSL connection to the server without the change.

    Edit the `ssl.client.props` file by doing the following:

    a. Modify com.ibm.security.useFIPS to be set to `true`.
    b. Add com.ibm.websphere.security.FIPSLevel=SP800-131 just below the useFips property.
    c. Change the com.ibm.ssl.protocol property to TLSv1.2.

What to do next

The SP800-131 standard strict mode requires that the SSL connection use the TLSv1.2 protocol. For a browser to access the administrative console or an application, the browser must support and first be configured to use the TLSv1.2 protocol.

gotcha: When enabling the security standards on a Network Deployment version of the product, the node and deployment manager can be in an incompatible protocol state. Since configuring the security standard requires the server to be restarted, it is recommended that all node agents and servers be stopped, leaving the deployment manager running. Once the configuration changes are made through the console, restart the deployment manager.

Manually sync the nodes with syncNode, and start the node agents and servers. To use syncNode, you might need to update the `ssl.client.props` file to communicate with the deployment manager.

Implementing tokens for security attribute propagation

As part of an extensible architecture, WebSphere Application Server enables you to implement your own tokens in which to propagate security attributes.

About this task

The following topics are covered in this section:

Procedure

- Implementing a custom propagation token
- Implementing a custom authorization token
- Implementing a custom a single sign-on token

- Implementing a custom authentication token
- Propagating a custom Java serializable object

Implementing a custom propagation token for security attribute propagation

This topic explains how you might create your own propagation token implementation, which is set on the running thread and propagated downstream.

About this task

The default propagation token usually is sufficient for propagating attributes that are not user-specific. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. You must deserialize the bytes at the target and add that information back on the thread by plugging in a custom login module into the inbound system login configurations. This task also might include encryption and decryption.

To implement a custom propagation token, you must complete the following steps:

Procedure

1. Write a custom implementation of the PropagationToken interface. Many different methods are available for implementing the PropagationToken interface. However, make sure that the methods that are required by the PropagationToken interface and the token interface are fully implemented.

   After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so that it has the required permissions for the server code.

   Tip: All of the token types that are defined by the propagation framework have similar interfaces. The token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the propagation token, might extend the abstract class and then most of the work is complete.

   To see an implementation of the propagation token, see “Example: com.ibm.wsspi.security.token.PropagationToken implementation” on page 948.
2. Add and receive the custom propagation token during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. You also can add the implementation from an application. However, to deserialize the information, you need to plug in a custom login module, which is discussed in “Propagating a custom Java serializable object for security attribute propagation” on page 975. The WSSecurityPropagationHelper class has APIs that are used to set a propagation token on the thread and to retrieve the token from the thread to make updates.

   The code sample in “Example: Custom propagation token login module” on page 952 shows how to determine if the login is an initial login or a propagation login. The difference between these login types is whether the WSTokenHolderCallback callback contains propagation data. If the callback does not contain propagation data, initialize a new custom propagation token implementation and set it on the thread. If the callback contains propagation data, look for your specific custom propagation token TokenHolder instance, convert the byte array back into your custom PropagationToken object, and set it back on the thread. The code sample shows both instances.

   You can add attributes any time your custom propagation token is added to the thread. If you add attributes between requests and the getUniqueId method changes, the Common Secure Interoperability Version 2 (CSIv2) client session is invalidated so that it can send the new information

downstream. Adding attributes between requests can affect performance. In many cases, you want the downstream requests to receive the new propagation token information.

To add the custom propagation token to the thread, call the WSSecurityPropagationHelper.addPropagationToken method. This call requires the WebSphereRuntimePerMission "setPropagationToken" Java 2 Security permission.

3. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module for receiving serialized versions of your custom propagation token You can also add this login module to any of the application logins where you might want to generate your custom propagation token on the thread during the login. Alternatively, you can generate the custom PropagationToken implementation from within your application. However, to deserialize it, you need to add the implementation to the system login modules.

Results

After completing these steps, you have implemented a custom PropagationToken.

Example: com.ibm.wsspi.security.token.PropagationToken implementation:

Use this file to see an example of a propagation token implementation. The following sample code does not extend an abstract class, but implements the com.ibm.wsspi.security.token.PropagationToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if considerable differences exist between how you handle the various token implementations.

For information on how to implement a custom propagation token, see “Implementing a custom propagation token for security attribute propagation” on page 947.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomPropagationTokenImpl implements com.ibm.wsspi.security.
   token.PropagationToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
   // 2 hours in millis, by default
  private static long expire_period_in_millis = 2*60*60*1000;
 private long counter = 0;

/**
 * The constructor that is used to create initial PropagationToken instance
 */

 public CustomAbstractTokenImpl ()
 {
  // set the token version
  addAttribute("version", "1");
  // set the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis() +
        expire_period_in_millis).toString());
 }

/**
 * The constructor that is used to deserialize the token bytes received
 * during a propagation login.
 */
 public CustomAbstractTokenImpl (byte[] token_bytes)
 {
  try
```

```
  {
       hashtable = (java.util.Hashtable) com.ibm.wsspi.security.token.
         WSOpaqueTokenHelper.deserialize(token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // if you set the expiration to 0, it does not expire
  if (expiration != 0)
  {
   // return if this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }

/**
 * Gets the expiration as a long type.
 * @return long
 */
 public long getExpiration()
 {
  // get the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // expiration is the first element (should only be one)
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
     // You can choose whether your token gets propagated. In some cases
     // you might want the token to be local only.
  return true;
 }

/**
 * Gets the principal that this token belongs to. If this token is an
 * authorization token, this principal string must match the authentication
 * token principal string or the message is rejected.
 * @return String
 */
 public String getPrincipal()
 {
  // It is not necessary for the PropagationToken to return a principal,
     // because it is not user-centric.
  return "";
 }

/**
 * Returns the unique identifier of the token based upon information that
 * the provider considers makes it a unique token. This identifier is used
 * for caching purposes and might be used in combination with other token
 * unique IDs that are part of the same Subject.
 *
 * This method should return null if you want the accessID of the user to
 * represent its uniqueness. This is the typical scenario.
 *
 * @return String
```

```
 */
 public String getUniqueID()
 {
     // If you want to propagate the changes to this token, change the
     // value that this unique ID returns whenever the token is changed.
     // Otherwise, CSIv2 uses an existing session when everything else is
     // the same. This getUniqueID is checked by CSIv2 to determine the
     // session lookup.
  return counter;
 }

/**
 * Gets the bytes to be sent across the wire. The information in the byte[]
 * needs to be enough to recreate the Token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
     // Do this if the object is set to read-only during login commit
     // because this guarantees that no new data is set.
    if (isReadOnly() && tokenBytes == null)
     tokenBytes = com.ibm.wsspi.security.token.WSOpaqueTokenHelper.
               serialize(hashtable);

     // You can deserialize this in the downstream login module using
         //  WSOpaqueTokenHelper.deserialize()
    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }

/**
 * Gets the name of the token, which is used to identify the byte[] in the
 * protocol message.
 * @return String
 */
 public String getName()
 {
  return this.getClass().getName();
 }

/**
 * Gets the version of the token as a short type. This code also is used
 * to identify the byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only. The implementation
 * needs to ensure that any setter methods check that this read-only flag has
 * been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is readonly
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
```

```
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name and value pair. Returns the previous values set
 * for the key, or returns null if the value is not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
 public String[] addAttribute(String key, String value)
 {
  // Gets the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // Increments the counter to change the uniqueID
   counter++;

   // Copies the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // Allocates a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // Adds the String to the current array list
   array.add(value);

   // Adds the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // Returns the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
 }


/**
 * Gets the list of all of the attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep clone of this token. This is typically used by the session
 * logic of the CSIv2 server to create a copy of the token as it exists in the
 * session.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.websphere.security.token.CustomPropagationTokenImpl deep_clone =
   new com.ibm.websphere.security.token.CustomPropagationTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
```

```
  }

      return deep_clone;
 }
}
```

Example: Custom propagation token login module:

This example shows how to determine if the login is an initial login or a propagation login.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options)
 {
  // (For more information on what to do during initialization, see
     // Developing custom login modules for a system login configuration for JAAS.)
 }

 public boolean login() throws LoginException
 {
  // (For more information on what to do during login, see
     // Developing custom login modules for a system login configuration for JAAS.)

  // Handles the WSTokenHolderCallback to see if this is an initial
     // or propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // handle exception
  }

  // Receives the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

  if (authzTokenList != null)
  {
   // Iterates through the list looking for your custom token
   for (int i=0; i<authzTokenList.size(); i++)
   {
    TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

    // Looks for the name and version of your custom PropagationToken implementation
    if (tokenHolder.getName().equals("
              com.ibm.websphere.security.token.CustomPropagationTokenImpl") &&
        tokenHolder.getVersion() == 1)
    {
     // Passes the bytes into your custom PropagationToken constructor
            //  to deserialize
     customPropToken = new
      com.ibm.websphere.security.token.CustomPropagationTokenImpl(tokenHolder.
                 getBytes());

    }
   }
  }
  else // This is not a propagation login. Create a new instance of
          // your PropagationToken implementation
  {
   // Adds a new custom propagation token. This is an initial login
   customPropToken = new com.ibm.websphere.security.token.CustomPropagationTokenImpl();

   // Adds any initial attributes
   if (customPropToken != null)
   {
    customPropToken.addAttribute("key1", "value1");
    customPropToken.addAttribute("key1", "value2");
    customPropToken.addAttribute("key2", "value1");
    customPropToken.addAttribute("key3", "something different");
   }
  }

  // Note: You can add the token to the thread during commit in case
     // something happens during the login.
 }

 public boolean commit() throws LoginException
 {
  // For more information on what to do during commit, see
     // Developing custom login modules for a system login configuration for JAAS.
  if (customPropToken != null)
  {
```

```
   // Sets the propagation token on the thread
   try
   {

    System.out.println(tc, "*** ADDED MY CUSTOM PROPAGATION TOKEN TO THE THREAD ***");
    // Prints out the values in the deserialized propagation token
    java.util.Enumeration keys = customPropToken.getAttributeNames();
    while (keys.hasMoreElements())
    {
     String key = (String) keys.nextElement();
     String[] list = (String[]) customPropToken.getAttributes(key);
     for (int k=0; k<list.length; k++)
     System.out.println("Key/Value: " + key + "/" + list[k]);
    }

    // This sets it on the thread using getName() + getVersion() as the key
    com.ibm.wsspi.security.token.WSSecurityPropagationHelper.addPropagationToken(
              customPropToken);
   }
   catch (Exception e)
   {
    // Handles exception
   }


   // Now you can verify that you have set it properly by trying to get
       // it back from the thread and print the values.
   try
   {
    // This gets the PropagationToken from the thread using getName()
          // and getVersion() parameters.
    com.ibm.wsspi.security.token.PropagationToken tempPropagationToken =
     com.ibm.wsspi.security.token.WSSecurityPropagationHelper.getPropagationToken
      ("com.ibm.websphere.security.token.CustomPropagationTokenImpl", 1);

    if  (tempPropagationToken != null)
    {
     System.out.println(tc, "*** RECEIVED MY CUSTOM PROPAGATION
               TOKEN FROM THE THREAD ***");
     // Prints out the values in the deserialized propagation token
     java.util.Enumeration keys = tempPropagationToken.getAttributeNames();
     while (keys.hasMoreElements())
     {
      String key = (String) keys.nextElement();
      String[] list = (String[]) tempPropagationToken.getAttributes(key);
      for (int k=0; k<list.length; k++)
      System.out.println("Key/Value: " + key + "/" + list[k]);
     }
    }
   }
   catch (Exception e)
   {
    // Handles exception
   }
  }
 }

 // Defines your login module variables
 com.ibm.wsspi.security.token.PropagationToken customPropToken = null;

}
```

Implementing a custom authorization token for security attribute propagation

This task explains how you might create your own AuthorizationToken implementation, which is set in the login Subject and propagated downstream.

About this task

The default AuthorizationToken usually is sufficient for propagating attributes that are user-specific. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. You must deserialize the bytes at the target and add that information back on the thread. This task also might include encryption and decryption.
- Affect the overall uniqueness of the Subject using the getUniqueID() application programming interface (API).

To implement a custom authorization token, you must complete the following steps:

Procedure

1. Write a custom implementation of the AuthorizationToken interface. There are many different methods for implementing the AuthorizationToken interface. However, make sure that the methods required by the AuthorizationToken interface and the token interface are fully implemented.

   After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so that it has the necessary permissions that are needed by the server code.

   Tip: All of the token types defined by the propagation framework have similar interfaces. Basically, the token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the AuthorizationToken, might extend the abstract class and then most of the work is completed.

   To see an implementation of AuthorizationToken, see "Example: com.ibm.wsspi.security.token.AuthorizationToken implementation"

2. Add and receive the custom AuthorizationToken during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. However, in order to deserialize the information, you must plug in a custom login module, which is discussed in "Propagating a custom Java serializable object for security attribute propagation" on page 975. After the object is instantiated in the login module, you can add the object to the Subject during the commit() method.

   If you only want to add information to the Subject to get propagated, see "Propagating a custom Java serializable object for security attribute propagation" on page 975. If you want to ensure that the information is propagated, want to do you own custom serialization, or want to specify the uniqueness for Subject caching purposes, then consider writing your own AuthorizationToken implementation.

   The code sample in "Example: custom AuthorizationToken login module" on page 958 shows how to determine if the login is an initial login or a propagation login. The difference between these login types is whether the WSTokenHolderCallback contains propagation data. If the callback does not contain propagation data, initialize a new custom AuthorizationToken implementation and set it into the Subject. If the callback contains propagation data, look for your specific custom AuthorizationToken TokenHolder instance, convert the byte[] back into your custom AuthorizationToken object, and set it back into the Subject. The code sample shows both instances.

   You can make your AuthorizationToken read-only in the commit phase of the login module. If you do not make the token read-only, then attributes can be added within your applications.

3. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule for receiving serialized versions of your custom authorization token.

   Because this login module relies on information in the sharedState added by the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule, add this login module after com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule. For information on how to add your custom login module to the existing login configurations, see Developing custom login modules for a system login configuration for JAAS.

Results

After completing these steps, you have implemented a custom AuthorizationToken.

Example: com.ibm.wsspi.security.token.AuthorizationToken implementation:

Use this file to see an example of a AuthorizationToken implementation. The following sample code does not extend an abstract class, but rather implements the com.ibm.wsspi.security.token.AuthorizationToken

interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if there are considerable differences between how you handle the various token implementations.

For information on how to implement a custom AuthorizationToken, see “Implementing a custom authorization token for security attribute propagation” on page 953.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomAuthorizationTokenImpl implements com.ibm.wsspi.security.
   token.AuthorizationToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
 private static long expire_period_in_millis = 2*60*60*1000;
  // 2 hours in millis, by default

/**
 * Constructor used to create initial AuthorizationToken instance
 */

 public CustomAuthorizationTokenImpl (String principal)
 {
  // Sets the principal in the token
  addAttribute("principal", principal);
  // Sets the token version
  addAttribute("version", "1");
  // Sets the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis() +
        expire_period_in_millis).toString());
 }

/**
 * Constructor used to deserialize the token bytes received during a
 * propagation login.
 */
 public CustomAuthorizationTokenImpl (byte[] token_bytes)
 {
  try
  {
   hashtable = (java.util.Hashtable) com.ibm.wsspi.security.token.
           WSOpaqueTokenHelper.deserialize(token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // if you set the expiration to 0, it does not expire
  if (expiration != 0)
  {
   // return if this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
```

```
 }

/**
 * Gets the expiration as a long.
 * @return long
 */
 public long getExpiration()
 {
  // Gets the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // The expiration is the first element. There should be only one expiration.
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
  // You can choose whether your token gets propagated. In some cases,
     // you might want it to be local only.
  return true;
 }

/**
 * Gets the principal that this Token belongs to. If this is an authorization token,
 * this principal string must match the authentication token principal string or the
 * message will be rejected.
 * @return String
 */
 public String getPrincipal()
 {
  // this might be any combination of attributes
  String[] principal = getAttributes("principal");

  if (principal != null && principal[0] != null)
  {
   return principal[0];
  }

  System.out.println("getExpiration: returning null");
  return null;
 }

/**
 * Returns a unique identifier of the token based upon the information that provider
 * considers makes this a unique token. This will be used for caching purposes
 * and might be used in combination with other token unique IDs that are part of
 * the same Subject.
 *
 * This method should return null if you want the accessID of the user to represent
 * uniqueness.  This is the typical scenario.
 *
 * @return String
 */
 public String getUniqueID()
 {
  // if you don't want to affect the cache lookup, just return NULL here.
  // return null;

  String cacheKeyForThisToken = "dynamic attributes";

  // if you do want to affect the cache lookup, return a string of
     // attributes that you want factored into the lookup.
  return cacheKeyForThisToken;
 }

/**
 * Gets the bytes to be sent across the wire. The information in the byte[]
 * needs to be enough to recreate the Token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
    // Do this if the object is set to read-only during login commit,
    // because this makes sure that no new data gets set.
    if (isReadOnly() && tokenBytes == null)
```

```
     tokenBytes = com.ibm.wsspi.security.token.WSOpaqueTokenHelper.
                serialize(hashtable);

    // You can deserialize this in the downstream login module using
          // WSOpaqueTokenHelper.deserialize()
    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }

/**
 * Gets the name of the token used to identify the byte[] in the protocol message.
 * @return String
 */
 public String getName()
 {
  return this.getClass().getName();
 }

/**
 * Gets the version of the token as an short. This also is used to identify the
 * byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only. The implementation
 * needs to ensure that any setter methods check that this flag has been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is read-only
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name and value pair. Returns the previous values set for key,
 * or null if not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
```

```
 public String[] addAttribute(String key, String value)
 {
  // Gets the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // Copies the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // Allocates a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // Adds the String to the current array list
   array.add(value);

   // Adds the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // Returns the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
 }


/**
 * Gets the list of all attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep copying of this token, if necessary.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.websphere.security.token.CustomAuthorizationTokenImpl deep_clone =
   new com.ibm.websphere.security.token.CustomAuthorizationTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }
}
```

Example: custom AuthorizationToken login module:

This file shows how to determine if the login is an initial login or a propagation login.

For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration for JAAS.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
  // Handles the WSTokenHolderCallback to see if this is an initial or
     // propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");
```

```
try
{
 callbackHandler.handle(callbacks);
}
catch (Exception e)
{
 // Handles exception
}

// Receives the ArrayList of TokenHolder objects (the serialized tokens)
List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

if (authzTokenList != null)
{
 // Iterates through the list looking for your custom token
 for (int i=0; i
 for (int i=0; i<authzTokenList.size(); i++)
 {
  TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

        // Looks for the name and version of your custom AuthorizationToken
        // implementation
  if (tokenHolder.getName().equals("com.ibm.websphere.security.token.
          CustomAuthorizationTokenImpl") &&
      tokenHolder.getVersion() == 1)
  {
         // Passes the bytes into your custom AuthorizationToken constructor
         // to deserialize
   customAuthzToken = new
    com.ibm.websphere.security.token.CustomAuthorizationTokenImpl(
                tokenHolder.getBytes());

  }
 }
}
else
   // This is not a propagation login. Create a new instance of your
   // AuthorizationToken implementation
{
     // Gets the prinicpal from the default AuthenticationToken. This must match
     // all tokens.
 defaultAuthToken = (com.ibm.wsspi.security.token.AuthenticationToken)
  sharedState.get(com.ibm.wsspi.security.auth.callback.Constants.WSAUTHTOKEN_KEY);
 String principal = defaultAuthToken.getPrincipal();

     // Adds a new custom authorization token. This is an initial login. Pass the
     // principal into the constructor
 customAuthzToken = new com.ibm.websphere.security.token.
         CustomAuthorizationTokenImpl(principal);

 // Adds any initial attributes
 if (customAuthzToken != null)
 {
  customAuthzToken.addAttribute("key1", "value1");
  customAuthzToken.addAttribute("key1", "value2");
  customAuthzToken.addAttribute("key2", "value1");
  customAuthzToken.addAttribute("key3", "something different");
 }
}

   // Note: You can add the token to the Subject during commit in case something
   // happens during the login.
}

public boolean commit() throws LoginException
{
 if (customAut  // (hzToken != null)
 {
  // sSets the customAuthzToken token into the Subject
  try
  {
   public final AuthorizationToken customAuthzTokenPriv = customAuthzToken;
         // Do this in a doPrivileged code block so that application code does not
         // need to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
   {
    public Object run()
    {
     try
     {
                // Adds the custom authorization token if it is not null
                //  and not already in the Subject
                            if ((customAuthzTokenPriv != null) &&
       (!subject.getPrivateCredentials().contains(customAuthzTokenPriv)))
      {
       subject.getPrivateCredentials().add(customAuthzTokenPriv);
      }
     }
     catch (Exception e)
     {
```

```
       throw new WSLoginFailedException (e.getMessage(), e);
      }

      return null;
     }
    });
   }
   catch (Exception e)
   {
    throw new WSLoginFailedException (e.getMessage(), e);
   }
  }
 }

 // Defines your login module variables
 com.ibm.wsspi.security.token.AuthorizationToken customAuthzToken = null;
 com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
 java.util.Map _sharedState = null;
}
```

Implementing a custom single sign-on token for security attribute propagation

You can create your own single sign-on token implementation. The single sign-on token implementation is set in the login Subject and added to the HTTP response as an HTTP cookie.

About this task

The cookie name is the concatenation of the SingleSignonToken.getName application programming interface (API) and the SingleSignonToken.getVersion API. There is no delimiter. When you add a single sign-on token to the Subject, it also gets propagated horizontally and downstream in case the Subject is used for other web requests. You must deserialize your custom single sign-on token when you receive it from a propagation login. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. Encrypt the information because it is out to the HTTP response and is available on the Internet. You must deserialize or decrypt the bytes at the target and add that information back into the Subject.
- Affect the overall uniqueness of the Subject using the getUniqueID API.

To implement a custom single sign-on token, complete the following steps:

Procedure

1. Write a custom implementation of the SingleSignonToken interface.

   Many different methods are available for implementing the SingleSignonToken interface. However, make sure the methods that are required by the SingleSignonToken interface and the token interface are fully implemented.

   After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so that it has the required permissions for the server code.

   Tip: All of the token types that are defined by the propagation framework have similar interfaces. Basically, the token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the single sign-on token, might extend the abstract class and then most of the work is complete.

   To see an implementation of the single sign-on token, see “Example: A com.ibm.wsspi.security.token.SingleSignonToken implementation” on page 961

2. Add and receive the custom single sign-on token during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system

login configurations. However, to deserialize the information, you need to plug in a custom login module, which is discussed in a subsequent step. After the object is instantiated in the login module, you can add it to the Subject during the commit method.

The code sample in “Example: A custom single sign-on token login module” on page 965, shows how to determine if the login is an initial login or a propagation login. The difference is whether the WSTokenHolderCallback callback contains propagation data. If the callback does not contain propagation data, initialize a new custom single sign-on token implementation and set it into the Subject. Also, look for the HTTP cookie from the HTTP request if the HTTP request object is available in the callback. You can get your custom single sign-on token both from a horizontal propagation login and from the HTTP request. However, it is recommended that you make the token available in both places because then the information arrives at any front-end application server, even if that server does not support propagation.

You can make your single sign-on token read-only in the commit phase of the login module. If you make the token read-only, additional attributes cannot be added within your applications.

Restriction:

- HTTP cookies have a size limitation. Size restrictions should be included in the documentation for your specific browser.
- The WebSphere Application Server runtime does not handle cookies that it does not generate, so this cookie is not used by the runtime.
- The SingleSignonToken object, when in the Subject, does affect the cache lookup of the Subject if you return something in the getUniqueID method.

3. Get the HTTP cookie from the HTTP request object during login or from an application. The sample code that is found in “Example: An HTTP cookie retrieval” on page 966 shows how you can retrieve the HTTP cookie from the HTTP request, decode the cookie so that it is back to your original bytes, and create your custom SingleSignonToken object from the bytes.
4. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule for receiving serialized versions of your custom propagation token. Because this login module relies on information in the sharedState state that is added by the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module, add this login module after the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module.

   For information on adding your custom login module into the existing login configurations, see Developing custom login modules for a system login configuration for JAAS.

Results

After completing these steps, you have implemented a custom single sign-on token.

Example: A com.ibm.wsspi.security.token.SingleSignonToken implementation:

Use this file to see an example of a single sign-on implementation. The following sample code does not extend an abstract class, but implements the com.ibm.wsspi.security.token.SingleSignonToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if considerable differences exist between how you handle the various token implementations.

For information on how to implement a custom single sign-on token, see “Implementing a custom single sign-on token for security attribute propagation” on page 960.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
```

```
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomSingleSignonTokenImpl implements com.ibm.wsspi.security.
   token.SingleSignonToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
   // 2 hours in millis, by default
 private static long expire_period_in_millis = 2*60*60*1000;

/**
 * Constructor used to create initial SingleSignonToken instance
 */

 public CustomSingleSignonTokenImpl (String principal)
 {
  // set the principal in the token
  addAttribute("principal", principal);
  // set the token version
  addAttribute("version", "1");
  // set the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis() +
        expire_period_in_millis).toString());
 }

/**
 * Constructor used to deserialize the token bytes received during a propagation login.
 */
 public CustomSingleSignonTokenImpl (byte[] token_bytes)
 {
  try
  {
   // you should implement a decryption algorithm to decrypt the cookie bytes
   hashtable = (java.util.Hashtable) some_decryption_algorithm (token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */

 public boolean isValid ()
 {
  long expiration = getExpiration();

  // if you set the expiration to 0, it does not expire
  if (expiration != 0)
  {
   // return if this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }

/**
 * Gets the expiration as a long.
 * @return long
 */
 public long getExpiration()
 {
  // get the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // expiration will always be the first element (should only be one)
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
```

```
 return 0;
}

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
public boolean isForwardable()
{
 // You can choose whether your token gets propagated or not, in some cases
    // you might want it to be local only.
 return true;
}

/**
 * Gets the principal that this Token belongs to. If this is an authorization token,
 * this principal string must match the authentication token principal string or the
 * message will be rejected.
 * @return String
 */
public String getPrincipal()
{
 // this could be any combination of attributes
 String[] principal = getAttributes("principal");

 if (principal != null && principal[0] != null)
 {
  return principal[0];
 }

 System.out.println("getExpiration: returning null");
 return null;
}

/**
 * Returns a unique identifier of the token based upon information the provider
 * considers makes this a unique token.  This will be used for caching purposes
 * and may be used in combination with other token unique IDs that are part of
 * the same Subject.
 *
 * This method should return null if you want the access ID of the user to represent
 * uniqueness.  This is the typical scenario.
 *
 * @return String
 */
public String getUniqueID()
{
 // this could be any combination of attributes
 return getPrincipal();
}

/**
 * Gets the bytes to be sent across the wire.  The information in the byte[]
 * needs to be enough to recreate the Token object at the target server.
 * @return byte[]
 */
public byte[] getBytes ()
{
 if (hashtable != null)
 {
  try
  {
   // do this if the object is set read-only during login commit,
   // since this guarantees no new data gets set.
   if (isReadOnly() && tokenBytes == null)
    tokenBytes = some_encryption_algorithm (hashtable);

   // you can deserialize the tokenBytes using a similiar decryption algorithm.
   return tokenBytes;
  }
  catch (Exception e)
  {
   e.printStackTrace();
   return null;
  }
 }

 System.out.println("getBytes: returning null");
 return null;
}

/**
 * Gets the name of the token, used to identify the byte[] in the protocol message.
 * @return String
 */
public String getName()
{
 return "myCookieName";
}
```

```
/**
 * Gets the version of the token as a short.  This is also used to identify the
 * byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only.  The implementation
 * needs to ensure any setter methods check that this has been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is readonly
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name/value pair.  Returns the previous values set for key,
 * or null if not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
 public String[] addAttribute(String key, String value)
 {
  // get the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // copy the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // allocate a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // add the String to the current array list
   array.add(value);

   // add the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // return the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
```

```
 }


/**
 * Gets the List of all attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep copying of this token, if necessary.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.websphere.security.token.CustomSingleSignonImpl deep_clone =
   new com.ibm.websphere.security.token.CustomSingleSignonTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }
}
```

Example: A custom single sign-on token login module:

This file shows how to determine if the login is an initial login or a propagation login.

For information on initialization and on what to do during login and commit, see Developing custom login modules for a system login configuration for JAAS.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
     // Handles the WSTokenHolderCallback to see if this is an initial or
     // propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // handle exception
  }

  // Receives the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

  if (authzTokenList != null)
  {
   // iterate through the list looking for your custom token
   for (int i=0; i
   for (int i=0; i<authzTokenList.size(); i++)
   {
    TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

    // Looks for the name and version of your custom SingleSignonToken
          // implementation
    if (tokenHolder.getName().equals("myCookieName")
              && tokenHolder.getVersion() == 1)
    {
     // Passes the bytes into your custom SingleSignonToken constructor
            // to deserialize
```

```
    customSSOToken = new
     com.ibm.websphere.security.token.CustomSingleSignonTokenImpl
                 (tokenHolder.getBytes());

   }
  }
 }
 else
        // This is not a propagation login. Create a new instance of your
        // SingleSignonToken implementation
 {
     // Gets the principal from the default SingleSignonToken. This principal
     //  must match all tokens.
  defaultAuthToken = (com.ibm.wsspi.security.token.AuthenticationToken)
   sharedState.get(com.ibm.wsspi.security.auth.callback.Constants.WSAUTHTOKEN_KEY);
  String principal = defaultAuthToken.getPrincipal();

  // Adds a new custom single sign-on (SSO) token. This is an initial login.
      //  Pass the principal into the constructor
  customSSOToken = new com.ibm.websphere.security.token.
         CustomSingleSignonTokenImpl(principal);

  // add any initial attributes
  if (customSSOToken != null)
  {
   customSSOToken.addAttribute("key1", "value1");
   customSSOToken.addAttribute("key1", "value2");
   customSSOToken.addAttribute("key2", "value1");
   customSSOToken.addAttribute("key3", "something different");
  }
 }

    // Note: You can add the token to the Subject during commit in case something
    // happens during the login.
}

public boolean commit() throws LoginException
{
 if (customSSOToken != null)
 {
  // Sets the customSSOToken token into the Subject
  try
  {
   public final SingleSignonToken customSSOTokenPriv = customSSOToken;
         // Do this in a doPrivileged code block so that application code does not
         // need to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
   {
    public Object run()
    {
     try
     {
      // Adds the custom SSO token if it is not null and
                //  not already in the Subject
                            if ((customSSOTokenPriv != null) &&
        (!subject.getPrivateCredentials().
                        contains(customSSOTokenPriv)))
      {
       subject.getPrivateCredentials().
                    add(customSSOTokenPriv);
      }
     }
     catch (Exception e)
     {
      throw new WSLoginFailedException (e.getMessage(), e);
     }

     return null;
    }
   });
  }
  catch (Exception e)
  {
   throw new WSLoginFailedException (e.getMessage(), e);
  }
 }
}

// Defines your login module variables
com.ibm.wsspi.security.token.SingleSignonToken customSSOToken = null;
com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
java.util.Map _sharedState = null;
}
```

Example: An HTTP cookie retrieval:

The following example shows you how to retrieve a cookie from an HTTP request, decode the cookie so that it is back to your original bytes, and create your custom SingleSignonToken object from the bytes. This example shows how to complete these steps from a login module. However, you also can complete these steps using a servlet.

For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration for JAAS.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
     // Handles the WSTokenHolderCallback to see if this is an
     // initial or propagation login.
  Callback callbacks[] = new Callback[2];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");
  callbacks[1] = new WSServletRequestCallback("HttpServletRequest: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // Handles the exception
  }

  // receive the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();
  javax.servlet.http.HttpServletRequest request =
         ((WSServletRequestCallback) callbacks[1]).getHttpServletRequest();

  if (request != null)
  {

   // Checks if the cookie is present
   javax.servlet.http.Cookie[] cookies = request.getCookies();
   String[] cookieStrings = getCookieValues (cookies, "myCookeName1");

   if (cookieStrings != null)
   {
    String cookieVal = null;
    for (int n=0;n<cookieStrings.length;n++)
    {
     cookieVal = cookieStrings[n];
     if (cookieVal.length()>0)
     {
               // Removes the cookie encoding from the cookie to get
               // your custom bytes
      byte[] cookieBytes =
       com.ibm.websphere.security.WSSecurityHelper.
                     convertCookieStringToBytes(cookieVal);
      customSSOToken =
       new com.ibm.websphere.security.token.
                     CustomSingleSignonTokenImpl(cookieBytes);

               // Now that you have your cookie from the request,
               // you can do something with it here, or add it
               // to the Subject in the commit() method for use later.
      if (debug || tc.isDebugEnabled())
      {
       System.out.println("*** GOT MY CUSTOM SSO TOKEN FROM
                     THE REQUEST ***");
      }
     }
    }
   }
  }

 }

 public boolean commit() throws LoginException
 {
  if (customSSOToken != null)
  {
   // Sets the customSSOToken token into the Subject
   try
   {
    public final SingleSignonToken customSSOTokenPriv = customSSOToken;
          // Do this in a doPrivileged code block so that application code does not
```

```
        // need to add additional permissions
    java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
    {
     public Object run()
     {
      try
      {
                 // Add the custom SSO token if it is not null and not
                 // already in the Subject
                              if ((customSSOTokenPriv != null) &&
         (!subject.getPrivateCredentials().
                       contains(customSSOTokenPriv)))
       {
        subject.getPrivateCredentials().add(customSSOTokenPriv);
       }
      }
      catch (Exception e)
      {
       throw new WSLoginFailedException (e.getMessage(), e);
      }

      return null;
     }
    });
   }
   catch (Exception e)
   {
    throw new WSLoginFailedException (e.getMessage(), e);
   }
  }
 }

 // Private method to get the specific cookie from the request
 private String[] getCookieValues (Cookie[] cookies, String hdrName)
 {
  Vector retValues = new Vector();
  int numMatches=0;
  if (cookies != null)
  {
   for (int i = 0; i < cookies.length; ++i)
   {
    if (hdrName.equals(cookies[i].getName()))
    {
     retValues.add(cookies[i].getValue());
     numMatches++;
     System.out.println(cookies[i].getValue());
    }
   }
  }

  if (retValues.size()>0)
   return (String[]) retValues.toArray(new String[numMatches]);
  else
   return null;
 }

 // Defines your login module variables
 com.ibm.wsspi.security.token.SingleSignonToken customSSOToken = null;
 com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
 java.util.Map _sharedState = null;
}
```

Implementing a custom authentication token for security attribute propagation

This topic explains how you might create your own authentication token implementation, which is set in the login Subject and propagated downstream.

About this task

With this implementation you can specify an authentication token that can be used by a custom login module or application. Consider writing your own implementation if you want to accomplish one of the following tasks:

- Isolate your attributes within your own implementation.
- Serialize the information using custom serialization. You must deserialize the bytes at the target and add that information back on the thread. This task also might include encryption and decryption.
- Affect the overall uniqueness of the Subject using the getUniqueID application programming interface (API).

Important: Custom authentication token implementations are not used by the security runtime in WebSphere Application Server to enforce authentication. WebSphere Application Security runtime uses this token in the following situations only:

- Call the getBytes method for serialization
- Call the getForwardable method to determine whether to serialize the authentication token.
- Call the getUniqueId method for uniqueness
- Call the getName and the getVersion methods for adding serialized bytes to the token holder that is sent downstream

All of the other uses are custom implementations.

To implement a custom authentication token, you must complete the following steps:

Procedure

1. Write a custom implementation of the AuthenticationToken interface. Many different methods are available for implementing the AuthenticationToken interface. However, make sure the methods that are required by the AuthenticationToken interface and the token interface are fully implemented. After you implement this interface, you can place it in the *app_server_root*/classes directory. Alternatively, you can place the class in any private directory. However, make sure that the WebSphere Application Server class loader can locate the class and that it is granted the appropriate permissions. You can add the Java archive (JAR) file or directory that contains this class into the server.policy file so the class has the necessary permissions required by the server code.

   Tip: All of the token types that are defined by the propagation framework have similar interfaces. The token types are marker interfaces that implement the com.ibm.wsspi.security.token.Token interface. This interface defines most of the methods. If you plan to implement more than one token type, consider creating an abstract class that implements the com.ibm.wsspi.security.token.Token interface. All of your token implementations, including the authentication token, might extend the abstract class and then most of the work is complete.

   To see an implementation of the AuthenticationToken interface, see “Example: A com.ibm.wsspi.security.token.AuthenticationToken implementation” on page 970.
2. Add and receive the custom authentication token during WebSphere Application Server logins. This task is typically accomplished by adding a custom login module to the various application and system login configurations. However, to deserialize the information you must plug in a custom login module. After the object is instantiated in the login module, you can add the object to the Subject during the commit method.

   If you only want to add information to the Subject to get propagated, see “Propagating a custom Java serializable object for security attribute propagation” on page 975. If you want to ensure that the information is propagated, do your own custom serialization, or specify the uniqueness for Subject caching purposes, consider writing your own authentication token implementation.

   The code sample in “Example: A custom authentication token login module” on page 974, shows how to determine if the login is an initial login or a propagation login. The difference between these login types is whether the WSTokenHolderCallback callback contains propagation data. If the callback does not contain propagation data, initialize a new custom authentication token implementation and set it into the Subject. If the callback contains propagation data, look for your specific custom authentication token TokenHolder instance, convert the byte array back into your custom AuthenticationToken object, and set it back into the Subject. The code sample shows both instances.

   You can make your authentication token read-only in the commit phase of the login module. If you do not make the token read-only, attributes can be added within your applications.
3. Add your custom login module to WebSphere Application Server system login configurations that already contain the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module for receiving serialized versions of your custom authorization token.

Because this login module relies on information in the shared state that is added by the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module, add this login module after the com.ibm.ws.security.server.lm.wsMapDefaultInboundLoginModule login module. For information on how to add your custom login module to the existing login configurations, see Developing custom login modules for a system login configuration for JAAS.

Results

After completing these steps, you have implemented a custom authentication token.

Example: A com.ibm.wsspi.security.token.AuthenticationToken implementation:

The following example illustrates an authentication token implementation. The following sample code does not extend an abstract class, but rather implements the com.ibm.wsspi.security.token.AuthenticationToken interface directly. You can implement the interface directly, but it might cause you to write duplicate code. However, you might choose to implement the interface directly if considerable differences exist between how you handle the various token implementations.

```
package com.ibm.websphere.security.token;

import com.ibm.websphere.security.WSSecurityException;
import com.ibm.websphere.security.auth.WSLoginFailedException;
import com.ibm.wsspi.security.token.*;
import com.ibm.websphere.security.WebSphereRuntimePermission;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.DataOutputStream;
import java.io.DataInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.OutputStream;
import java.io.InputStream;
import java.util.ArrayList;

public class CustomAuthenticationTokenImpl implements com.ibm.wsspi.security.
   token.AuthenticationToken
{
 private java.util.Hashtable hashtable = new java.util.Hashtable();
 private byte[] tokenBytes = null;
  // 2 hours in millis, by default
   private static long expire_period_in_millis = 2*60*60*1000;
 private String oidName = "your_oid_name";
  // This string can really be anything if you do not want to use an OID.

/**
 * Constructor used to create initial AuthenticationToken instance
 */
 public CustomAuthenticationTokenImpl (String principal)
 {
  // Sets the principal in the token
  addAttribute("principal", principal);
  // Sets the token version
  addAttribute("version", "1");
  // Sets the token expiration
  addAttribute("expiration", new Long(System.currentTimeMillis()
        + expire_period_in_millis).toString());
 }

/**
 * Constructor used to deserialize the token bytes received during a
 * propagation login.
 */
 public CustomAuthenticationTokenImpl (byte[] token_bytes)
 {
  try
  {
       // The data in token_bytes should be signed and encrypted if the
       // hashtable is acting as an authentication token.
   hashtable = (java.util.Hashtable) custom_decryption_algorithm (token_bytes);
  }
  catch (Exception e)
  {
   e.printStackTrace();
  }
 }

/**
 * Validates the token including expiration, signature, and so on.
 * @return boolean
 */
```

```
 public boolean isValid ()
 {
  long expiration = getExpiration();

  // If you set the expiration to 0, the token does not expire
  if (expiration != 0)
  {
   // Returns a response that identifies whether this token is still valid
   long current_time = System.currentTimeMillis();

   boolean valid = ((current_time < expiration) ? true : false);
   System.out.println("isValid: returning " + valid);
   return valid;
  }
  else
  {
   System.out.println("isValid: returning true by default");
   return true;
  }
 }

/**
 * Gets the expiration as a long type.
 * @return long
 */
 public long getExpiration()
 {
  // Gets the expiration value from the hashtable
  String[] expiration = getAttributes("expiration");

  if (expiration != null && expiration[0] != null)
  {
   // The expiration is the first element and there should only be one expiration
   System.out.println("getExpiration: returning " + expiration[0]);
   return new Long(expiration[0]).longValue();
  }

  System.out.println("getExpiration: returning 0");
  return 0;
 }

/**
 * Returns if this token should be forwarded/propagated downstream.
 * @return boolean
 */
 public boolean isForwardable()
 {
     // You can choose whether your token gets propagated. In some cases
     // you might want it to be local only.
  return true;
 }

/**
 * Gets the principal to which this token belongs. If this is an
 * authorization token, this principal string must match the
 * authentication token principal string or the message is rejected.
 * @return String
 */
 public String getPrincipal()
 {
  // This value might be any combination of attributes
  String[] principal = getAttributes("principal");

  if (principal != null && principal[0] != null)
  {
   return principal[0];
  }

  System.out.println("getExpiration: returning null");
  return null;
 }

/**
 * Returns a unique identifier of the token based upon information the provider
 * considers makes this a unique token. This identifier is used for caching purposes
 * and can be used in combination with other token unique IDs that are part of
 * the same Subject.
 *
 * This method should return null if you want the accessID of the user to represent
 * uniqueness.  This is the typical scenario.
 *
 * @return String
 */
 public String getUniqueID()
 {
     // If you do not want to affect the cache lookup, just return NULL here.
     return null;

  String cacheKeyForThisToken = "dynamic attributes";
```

```
    // If you do want to affect the cache lookup, return a string of
    // attributes that you want factored into the lookup.
  return cacheKeyForThisToken;
 }

/**
 * Gets the bytes to be sent across the wire. The information in the byte[]
 * needs to be enough to recreate the token object at the target server.
 * @return byte[]
 */
 public byte[] getBytes ()
 {
  if (hashtable != null)
  {
   try
   {
          // Do this if the object is set read-only during login commit
          // because this ensures that new data is not set.
    if (isReadOnly() && tokenBytes == null)
     tokenBytes = custom_encryption_algorithm (hashtable);

    return tokenBytes;
   }
   catch (Exception e)
   {
    e.printStackTrace();
    return null;
   }
  }

  System.out.println("getBytes: returning null");
  return null;
 }

/**
 * Gets the name of the token, which is used to identify the byte[] in the
 * protocol message.
 * @return String
 */
 public String getName()
 {
  return oidName;
 }

/**
 * Gets the version of the token as a short type.  This also is used
 * to identify the byte[] in the protocol message.
 * @return short
 */
 public short getVersion()
 {
  String[] version = getAttributes("version");

  if (version != null && version[0] != null)
   return new Short(version[0]).shortValue();

  System.out.println("getVersion: returning default of 1");
  return 1;
    }

/**
 * When called, the token becomes irreversibly read-only.  The implementation
 * needs to ensure that any set methods check that this state has been set.
 */
 public void setReadOnly()
 {
  addAttribute("readonly", "true");
 }

/**
 * Called internally to see if the token is read-only
 */
 private boolean isReadOnly()
 {
  String[] readonly = getAttributes("readonly");

  if (readonly != null && readonly[0] != null)
   return new Boolean(readonly[0]).booleanValue();

  System.out.println("isReadOnly: returning default of false");
  return false;
 }

/**
 * Gets the attribute value based on the named value.
 * @param String key
 * @return String[]
 */
 public String[] getAttributes(String key)
```

```
 {
  ArrayList array = (ArrayList) hashtable.get(key);

  if (array != null && array.size() > 0)
  {
   return (String[]) array.toArray(new String[0]);
  }

  return null;
 }

/**
 * Sets the attribute name/value pair.  Returns the previous values set for key,
 * or null if not previously set.
 * @param String key
 * @param String value
 * @returns String[];
 */
 public String[] addAttribute(String key, String value)
 {
  // Gets the current value for the key
  ArrayList array = (ArrayList) hashtable.get(key);

  if (!isReadOnly())
  {
   // Copies the ArrayList to a String[] as it currently exists
   String[] old_array = null;
   if (array != null && array.size() > 0)
    old_array = (String[]) array.toArray(new String[0]);

   // Allocates a new ArrayList if one was not found
   if (array == null)
    array = new ArrayList();

   // Adds the String to the current array list
   array.add(value);

   // Adds the current ArrayList to the Hashtable
   hashtable.put(key, array);

   // Returns the old array
   return old_array;
  }

  return (String[]) array.toArray(new String[0]);
 }


/**
 * Gets the list of all attribute names present in the token.
 * @return java.util.Enumeration
 */
 public java.util.Enumeration getAttributeNames()
 {
  return hashtable.keys();
 }

/**
 * Returns a deep copying of this token, if necessary.
 * @return Object
 */
 public Object clone()
 {
  com.ibm.wsspi.security.token.AuthenticationToken deep_clone =
   new com.ibm.websphere.security.token.CustomAuthenticationTokenImpl();

  java.util.Enumeration keys = getAttributeNames();

  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();

   String[] list = (String[]) getAttributes(key);

   for (int i=0; i<list.length; i++)
    deep_clone.addAttribute(key, list[i]);
  }

      return deep_clone;
 }

/**
 * This method returns true if this token is storing a user ID and password
 * instead of a token.
 * @return boolean
 */
 public boolean isBasicAuth()
```

```
 {
  return false;
 }
}
```

Example: A custom authentication token login module:

This examples shows how to determine if the login is an initial login or a propagation login.

For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration for JAAS.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
    Map sharedState, Map options)
 {
  _sharedState = sharedState;
 }

 public boolean login() throws LoginException
 {
  // Handles the WSTokenHolderCallback to see if this is an initial or
     // propagation login.
  Callback callbacks[] = new Callback[1];
  callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

  try
  {
   callbackHandler.handle(callbacks);
  }
  catch (Exception e)
  {
   // Handles exception
  }

  // Receives the ArrayList of TokenHolder objects (the serialized tokens)
  List authzTokenList = ((WSTokenHolderCallback) callbacks[0]).getTokenHolderList();

  if (authzTokenList != null)
  {
   // Iterates through the list looking for your custom token
   for (int i=0; i<authzTokenList.size(); i++)
   {
    TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

          // Looks for the name and version of your custom AuthenticationToken
          // implementation
    if (tokenHolder.getName().equals("your_oid_name") && tokenHolder.getVersion() == 1)
    {
            // Passes the bytes into your custom AuthenticationToken constructor
            // to deserialize
     customAuthzToken = new
      com.ibm.websphere.security.token.
               CustomAuthenticationTokenImpl(tokenHolder.getBytes());

    }
   }
  }
  else
          // This is not a propagation login. Create a new instance of your
          // AuthenticationToken implementation
  {
        //  Gets the principal from the default AuthenticationToken. This principal
        //  should match all default tokens.
        //  Note: WebSphere Application Server runtime only enforces this for
        //  default tokens. Thus, you can choose
        //  to do this for custom tokens, but it is not required.
   defaultAuthToken = (com.ibm.wsspi.security.token.AuthenticationToken)
    sharedState.get(com.ibm.wsspi.security.auth.callback.Constants.WSAUTHTOKEN_KEY);
   String principal = defaultAuthToken.getPrincipal();

       // Adds a new custom authentication token. This is an initial login. Pass
       // the principal into the constructor
   customAuthToken = new com.ibm.websphere.security.token.
          CustomAuthenticationTokenImpl(principal);

   // Adds any initial attributes
   if (customAuthToken != null)
   {
    customAuthToken.addAttribute("key1", "value1");
    customAuthToken.addAttribute("key1", "value2");
    customAuthToken.addAttribute("key2", "value1");
    customAuthToken.addAttribute("key3", "something different");
   }
  }
```

```
    // Note: You can add the token to the Subject during commit in case
    // something happens during the login.
}

public boolean commit() throws LoginException
{
 if (customAuthToken != null)
 {
  // Sets the customAuthToken token into the Subject
  try
  {
   private final AuthenticationToken customAuthTokenPriv = customAuthToken;
         // Do this in a doPrivileged code block so that application code does
         // not need to add additional permissions
   java.security.AccessController.doPrivileged(new java.security.PrivilegedAction()
   {
    public Object run()
    {
     try
     {
                 // Adds the custom Authentication token if it is not
                 // null and not already in the Subject
                              if ((customAuthTokenPriv != null) &&
        (!subject.getPrivateCredentials().
                        contains(customAuthTokenPriv)))
      {
       subject.getPrivateCredentials().add(customAuthTokenPriv);
      }
     }
     catch (Exception e)
     {
      throw new WSLoginFailedException (e.getMessage(), e);
     }

     return null;
    }
   });
  }
  catch (Exception e)
  {
   throw new WSLoginFailedException (e.getMessage(), e);
  }
 }
}

// Defines your login module variables
com.ibm.wsspi.security.token.AuthenticationToken customAuthToken = null;
com.ibm.wsspi.security.token.AuthenticationToken defaultAuthToken = null;
java.util.Map _sharedState = null;
}
```

Propagating a custom Java serializable object for security attribute propagation

This document describes how to add an object into the Subject from a login module and describes other infrastructure considerations to make sure that the Java object gets propagated.

Before you begin

Prior to completing this task, verify that security propagation is enabled in the administrative console.

About this task

With security attribute propagation enabled, you can propagate data either horizontally with single sign-on (SSO) enabled or downstream using Common Secure Interoperability Version 2 (CSIv2). When a login occurs, either through an application login configuration or a system login configuration, a custom login module can be plugged in to add Java serialized objects into the Subject during login. This document describes how to add an object into the Subject from a login module and describes other infrastructure considerations to make sure that the Java object gets propagated.

Procedure

1. Add your custom Java object into the Subject from a custom login module. A two-phase process exists for each Java Authentication and Authorization Service (JAAS) login module. WebSphere Application Server completes the following processes for each login module present in the configuration:

login method

In this step, the login configuration callbacks are analyzed, if necessary, and the new objects or credentials are created.

commit method

In this step, the objects or credentials that are created during login are added into the Subject.

After a custom Java object is added into the Subject, WebSphere Application Server serializes the object on the sending server, deserializes the object on the receiving server, and adds the object back into the Subject downstream. However, some requirements exist for this process to occur successfully. For more information on the JAAS programming model, see the JAAS information provided in the Security: Resources for learning article.

Important: Whenever you plug a custom login module into the login infrastructure of WebSphere Application Server, make sure that the code is trusted. When you put the classes together in a Java archive (JAR) file and add the file to the *app_server_root*/lib/ext/ directory, the login module has Java 2 Security AllPermissions permissions . It is recommended that you add your login module and other infrastructure classes into any private directory. However, you must modify the *profile_root*/properties/server.policy file to make sure that your private directory, Java archive (JAR) file, or both have the permissions required to run the application programming interfaces (API) that are called from the login module. Because the login module might be run after the application code on the call stack, you might add doPrivileged code so that you do not need to add additional properties to your applications.

The following code sample shows how to add doPrivileged code. For information on what to do during initialization, login and commit, see Developing custom login modules for a system login configuration for JAAS.

```
public customLoginModule()
{
 public void initialize(Subject subject, CallbackHandler callbackHandler,
     Map sharedState, Map options)
 {

 }

 public boolean login() throws LoginException
 {
     // Construct callback for the WSTokenHolderCallback so that you
     // can determine if
     // your custom object has propagated
    Callback callbacks[] = new Callback[1];
    callbacks[0] = new WSTokenHolderCallback("Authz Token List: ");

    try
    {
         _callbackHandler.handle(callbacks);
    }
    catch (Exception e)
    {
    throw new LoginException (e.getLocalizedMessage());
    }

    // Checks to see if any information is propagated into this login
    List authzTokenList = ((WSTokenHolderCallback) callbacks[1]).
            getTokenHolderList();

    if (authzTokenList != null)
    {
        for (int i = 0; i< authzTokenList.size(); i++)
             {
            TokenHolder tokenHolder = (TokenHolder)authzTokenList.get(i);

                  //  Look for your custom object. Make sure you use
                  //  "startsWith"because there is some data appended
                  //  to the end of the name indicating in which Subject
                  //  Set it belongs. Example from getName():
                  //  "com.acme.CustomObject (1)". The class name is
                  //  generated at the sending side by calling the
                  //  object.getClass().getName() method. If this object
                  //  is deserialized by WebSphere Application Server,
                  //  then return it and you do not need to add it here.
                  //  Otherwise, you can add it below.
                  //  Note: If your class appears in this list and does
                  //  not use custom serialization (for example, an
                  //  implementation of the Token interface described in
```

```
                //  the Propagation Token Framework), then WebSphere
                //  Application  Server automatically deserializes the
                //  Java object for you. You might just return here if
                //  it is found in the list.

            if (tokenHolder.getName().startsWith("com.acme.CustomObject"))
           return true;
       }
    }
       //  If you get to this point, then your custom object has not propagated
        myCustomObject = new com.acme.CustomObject();
        myCustomObject.put("mykey", "mydata");
}

public boolean commit() throws LoginException
{
 try
 {
      // Assigns a reference to a final variable so it can be used in
      // the doPrivileged block
  final com.acme.CustomObject myCustomObjectFinal = myCustomObject;
  // Prevents your applications from needing a JAAS getPrivateCredential
      // permission.
  java.security.AccessController.doPrivileged(new java.security.
          PrivilegedExceptionAction()
  {
   public Object run() throws java.lang.Exception
   {
           // Try not to add a null object to the Subject or an object
           // that already exists.
    if (myCustomObjectFinal != null && !subject.getPrivateCredentials().
              contains(myCustomObjectFinal))
    {
             // This call requires a special Java 2 Security permission,
             // see the JAAS application programming interface (API)
             // documentation.
     subject.getPrivateCredentials().add(myCustomObjectFinal);
    }
    return null;
   }
  });
 }
 catch (java.security.PrivilegedActionException e)
 {
  // Wraps the exception in a WSLoginFailedException
  java.lang.Throwable myException = e.getException();
  throw new WSLoginFailedException (myException.getMessage(), myException);
 }
}

// Defines your login module variables
com.acme.CustomObject myCustomObject = null;
}
```

2. Verify that your custom Java class implements the java.io.Serializable interface. An object that is added to the Subject must be serialized if you want the object to propagate. For example, the object must implement the java.io.Serializable interface. If the object is not serialized, the request does not fail, but the object does not propagate. To make sure an object that is added to the Subject is propagated, implement one of the token interfaces that is defined in topics about security attribute propagation or add attributes to one of the following existing default token implementations:

   AuthorizationToken
   Add attributes if they are user-specific.

   PropagationToken
   Add attributes that are specific to an invocation.

   If you are careful adding custom objects and follow all the steps to make sure that WebSphere Application Server can serialize and deserialize the object at each hop, then it is sufficient to use custom Java objects only.

3. Verify that your custom Java class exists on all of the systems that might receive the request.

   When you add a custom object into the Subject and expect WebSphere Application Server to propagate the object, put the class definitions together in a Java archive (JAR) file and add the file to the *app_server_root*/lib/ext/ directory on all of the nodes where serialization or deserialization might occur. Also, verify that the Java class versions are the same.

4. Verify that your custom login module is configured in all of the login configurations used in your environment where you need to add your custom object during a login. Any login configuration that interacts with WebSphere Application Server generates a Subject that might be propagated outbound

for an Enterprise JavaBeans (EJB) request. If you want WebSphere Application Server to propagate a custom object in all cases, make sure that the custom login module is added to every login configuration that is used in your environment. For more information, see Developing custom login modules for a system login configuration for JAAS.

5. Verify that security attribute propagation is enabled on all of the downstream servers that receive the propagated information. When an EJB request is sent to a downstream server and security attribute propagation is disabled on that server, only the authentication token is sent for backwards compatibility. Therefore, you must review the configuration to verify that propagation is enabled in all of the cells that might receive requests. You must check several places in the administrative console to make sure propagation is fully enabled.
6. Add any custom objects to the propagation exclude list that you do not want to propagate. You can configure a property to exclude the propagation of objects that match specific class names, package names, or both. For example, you can have a custom object that is related to a specific process. If the object is propagated, it does not contain valid information. You must tell WebSphere Application Server not to propagate this object. Complete the following steps to specify the object in the propagation exclude list, using the administrative console:
   a. Click **Security > Global security > Custom properties > New**.
   b. Add `com.ibm.ws.security.propagationExcludeList` in the **Name** field.
   c. Add the name of the custom object in the **Value** field. You can add a list of custom objects to the propagation exclude list, separated by a colon (:). For example, you might enter `com.acme.CustomLocalObject:com.acme.private.*`. You can enter a class name such as `com.acme.CustomLocalObject` or a package name such as `com.acme.private.*`. In this example, WebSphere Application Server does not propagate any class that equals `com.acme.CustomLocalObject` or begins with `com.acme.private.`

      Although you can add custom objects to the propagation exclude list, you must be aware of a side effect. WebSphere Application Server stores the opaque token, or the serialized Subject contents, in a local cache for the life of the single sign-on (SSO) token. The life of the SSO token, which has a default of two hours, is configured in the SSO properties on the administrative console. The information that is added to the opaque token includes only the objects not in the exclude list.

      Ensure that your SSO token timeout value is greater that the authentication cache timeout value. To modify the authentication cache, see the documentation about the authentication cache settings.

Results

As a result of this task, custom Java serializable objects are propagated horizontally or downstream. For more information on the differences between horizontal and downstream propagation, see topics about security attribute propagation or add attributes to one of the following existing default token implementations:.

Developing a custom interceptor for trust associations

You can define the interceptor class method that you want to use. WebSphere Application Server supports two trust association interceptor interfaces: com.ibm.wsspi.security.TrustAssociationInterceptor and com.ibm.wsspi.security.tai.TrustAssociationInterceptor.

Before you begin

If you are using a third party reverse proxy server other than Tivoli® WebSEAL, you must provide an implementation class for the product interceptor interface for your proxy server. This article describes the com.ibm.wsspi.security.TrustAssociationInterceptor.java interface that you must implement.

Note: The Trust Association Interceptor (TAI) interface (com.ibm.wsspi.security.tai.TrustAssociationInterceptor) supports several new features and is different from the existing com.ibm.wsspi.security.TrustAssociationInterceptor interface.

Procedure

1. Define the interceptor class method. WebSphere Application Server provides the interceptor Java interface, com.ibm.wsspi.security.TrustAssociationInterceptor, which defines the following methods:
   - **public boolean isTargetInterceptor(HttpServletRequest req)** creates WebTrustAssociationException;.

     The isTargetInterceptor method determines whether the request originated with the proxy server associated with the interceptor. The implementation code must examine the incoming request object and determine if the proxy server forwarding the request is a valid proxy server for this interceptor. The result of this method determines whether the interceptor processes the request or not.
   - **public void validateEstablishedTrust (HttpServletRequest req)** creates WebTrustAssociationException;.

     The validateEstablishedTrust method determines if the proxy server from which the request originated is trusted or not. This method is called after the isTargetInterceptor method. The implementation code must authenticate the proxy server. The authentication mechanism is proxy-server specific. For example, in the product implementation for the WebSEAL server, this method retrieves the basic authentication information from the HTTP header and validates the information against the user registry used by WebSphere Application Server. If the credentials are invalid, the code creates the WebTrustAssociationException, indicating that the proxy server is not trusted and the request is to be denied.
   - **public String getAuthenticatedUsername(HttpServletRequest req)** creates WebTrustAssociationException;.

     The getAuthenticatedUsername method is called after trust is established between the proxy server and WebSphere Application Server. The product has accepted the proxy server authentication of the request and must now authorize the request. To authorize the request, the name of the original requestor must be subjected to an authorization policy to determine if the requestor has the necessary privilege. The implementation code for this method must extract the user name from the HTTP request header and determine if that user is entitled to the requested resource. For example, in the product implementation for the WebSEAL server, the method looks for an iv-user attribute in the HTTP request header and extracts the user ID associated with it for authorization.
2. Configuring the interceptor. To make an interceptor configurable, the interceptor must extend com.ibm.wsspi.security.WebSphereBaseTrustAssociationInterceptor. Implement the following methods:
   public int init (java.util.Properties props);

   The init(Properties) method accepts a java.util.Properties object, which contains the set of properties required to initialize the interceptor. All the properties set for an interceptor (by using the **Custom Properties** link for that interceptor or using scripting) is sent to this method. The interceptor then can use these properties to initialize itself. For example, in the product implementation for the WebSEAL server, this method reads the hosts and ports so that a request coming in can be verified to originate from trusted hosts and ports. A return value of **0** implies that the interceptor initialization is successful. Any other value implies that the initialization is not successful and the interceptor is ignored.

   Applicability of the following list

   If a previous implementation of the trust association interceptor returns a different error status you can either change your implementation to match the expectations or make one of the following changes:
   - Add the com.ibm.wsspi.security.trustassociation.initStatus property in the trust association interceptor custom properties. Set the property to the value that indicates that the interceptor is successfully initialized. All of the other possible values imply failure. In case of failure, the corresponding trust association interceptor is not used.
   - Add the com.ibm.wsspi.security.trustassociation.ignoreInitStatus property in the trust association interceptor custom properties. Set the value of this property to **true**, which tells WebSphere Application Server to ignore the status of this method. If you add this property to the custom properties, WebSphere Application Server does not check the return status, which is similar to previous versions of WebSphere Application Server.

public void cleanup ();
This method is called when the application server is stopped. It is used to prepare the interceptor for termination.

public void setVersion (String s);
This method is optional. The method is used to set the version and is for informational purpose only. The default value is `Unspecified`.

You must configure the following methods implemented by the custom interceptor implementation. **This listing only shows the methods and does not include any implementation.**

```
*********************************************************************
import java.util.*;
import javax.servlet.http.HttpServletRequest;
import com.ibm.websphere.security.*;

public class myTAIImpl extends WebSphereBaseTrustAssociationInterceptor
      implements TrustAssociationInterceptor
{

      public myTAIImpl ()
      {
      }


      public boolean isTargetInterceptor (HttpServletRequest req)
          throws WebTrustAssociationException
      {

           //return true if this is the target interceptor, else return false.
      }


      public TAIResult negotiateValidateandEstablishTrust (HttpServletRequest req, HttpServletResponse res)
           throws WebTrustAssociationFailedException
      {
           //validate the request and establish trust.
           //create and return the TAIResult
public int initialize (Properties props)
      {
           //initialize the implementation. If successful return 0, else return 1.
      }

     public String getVersion()
     {
        //Return version
     }

   public String getType()
     {
        //Return type
     }

      public void cleanup ()
      {
           //Cleanup code.

      }


}
```

Note: If the init(Properties) method is implemented as described previously in your custom interceptor, this note does not apply to your implementation, and you can move on to the next step. Previous versions of com.ibm.wsspi.security.WebSphereBaseTrustAssociationInterceptor include the public int init (String propsfile) method. This method is no longer required since the interceptor properties are not read from a file. The properties are now entered in the administrative console **Custom Properties** link of the interceptor using the administrative console or scripts. These properties then are made available to your implementation in the

init(Properties) method. However, for backward compatibility, the init(String) method still is supported. The init(String) method is called by the default implementation of init(Properties) as shown in the following example.

```
// Default implementation of init(Properties props) method. A Custom
   // implementation should override this.
   public int init (java.util.Properties props)
   {
      String type =
       props.getProperty("com.ibm.wsspi.security.trustassociation.types");
      String classfile=
       props.getProperty("com.ibm.wsspi.security.trustassociation."
       +type+".config");
      if (classfile != null && classfile.length() > 0 ) {
         return init(classfile);
      } else {
         return -1;
      }
   }
```

Change your implementation to implement the init(Properties) method instead of relying on init(String propsfile) method. As shown in the previous example, this default implementation reads the properties to load the property file. The com.ibm.wsspi.security.trustassociation.types property gets the file containing the properties by concatenating `.config` to its value.

Note: The init(String) method still works if you want to use it instead of implementing the init(Properties) method. The only requirement is that the file name containing the custom trust association properties should now be entered using the **Custom Properties** link of the interceptor in the administrative console or by using scripts. You can enter the property using *either* of the following methods. The first method is used for backward compatibility with previous versions of WebSphere Application Server.

Method 1:

The same property names used in the previous release are used to obtain the file name. The file name is obtained by concatenating the `.config` to the com.ibm.wsspi.security.trustassociation.types property value.

If the file name is called `myTAI.properties` and is located in the */properties* directory, set the following properties:

- com.ibm.wsspi.security.trustassociation.types = myTAItype
- com.ibm.wsspi.security.trustassociation.myTAItype.config = *app_server_root*/myTAI.properties

Method 2:

You can set the com.ibm.wsspi.security.trustassociation.initPropsFile property in the trust association custom properties to the location of the file. For example, set the following property:

```
com.ibm.wsspi.security.trustassociation.initPropsFile=
app_server_root/myTAI.properties
```

Type the previous code as one continuous line.

The location of the properties file is fully qualified (for example,*app_server_root*/`myTAI.properties`). Because the location can be different in a WebSphere Application Server, Network Deployment environment, use variables such as `${USER_INSTALL_ROOT}` to refer to the WebSphere Application Server installation directory. For example, if the file name is called `myTAI.properties`, and it is located in the `/properties` directory, then set the following properties:

- com.ibm.wsspi.security.trustassociation.types = myTAItype
- com.ibm.wsspi.security.trustassociation.myTAItype.config = *app_server_root*/myTAI.properties

3. Compile the implementation once you have implemented it. For example, *app_server_root*`/java/bin/javac -classpath` *install_root*`/plugins/com.ibm.ws.runtime.jar;<install_root>/dev/JavaEE/j2ee.jar myTAIImpl.java`
   a. Identify the trust association interceptor class file for use when the server is restarted. Place the file either at the*app_server_root/classes* directory OR use the Java Virtual Machine (JVM) system property, `-Dws.ext.dirs` to specify where the file resides.

b. Restart all the servers.

4. Delete the default WebSEAL interceptor in the administrative console and click **New** to add your custom interceptor. Verify that the class name is dot separated and appears in the class path.
5. Click the **Custom Properties** link to add additional properties that are required to initialize the custom interceptor. These properties are passed to the init(Properties) method of your implementation when it extends the com.ibm.wsspi.security.WebSphereBaseTrustAssociationInterceptor as described in the previous step.
6. Save and synchronize (if applicable) the configuration.
7. Restart the servers for the custom interceptor to take effect.

Example

Refer to the Security: Resources for Learning article for a reference to an example of a custom interceptor.

Trust association interceptor support for Subject creation

The trust association interceptor (TAI) com.ibm.wsspi.security.tai.TrustAssociationInterceptor interface supports several features that are different from the existing com.ibm.websphere.security.TrustAssociationInterceptor interface.

The TAI interface supports a multiphase, negotiated authentication process. For example, some systems require a challenge response protocol back to the client. The two key methods in this interface are:

Key method name

public boolean isTargetInterceptor (HttpServletRequest req)

The isTargetInterceptor method determines whether the request originated with the proxy server that is associated with the interceptor. The implementation code must examine the incoming request object and determine if the proxy server that forwards the request is a valid proxy server for this interceptor. The result of this method determines whether the interceptor processes the request.

Method result

A `true` value tells WebSphere Application Server to have the TAI handle the request.

A `false` value, tells WebSphere Application Server to ignore the TAI.

Key method name

public TAIResult negotiateValidateandEstablishTrust (HttpServletRequest req, HttpServletResponse res)

The negotiateValidateandEstablishTrust method determines whether to trust the proxy server from which the request originated. The implementation code must authenticate the proxy server. The authentication mechanism is proxy-server specific. For example, in the product implementation for the WebSEAL server, this method retrieves the basic authentication information from the HTTP header and validates the information against the user registry that WebSphere Application Serve uses. If the credentials are not valid, the code creates the WebTrustAssociationException exception, which indicates that the proxy server is not trusted and the request is denied. If the credentials are valid, the code returns a TAIResult result, which indicates the status of the request processing with the client identity (Subject and principal name) to use for authorizing the web resource.

Method result

Returns a TAIResult result, which indicates the status of the request processing. You can query the Request object and modify the Response object can be modified.

The TAIResult class has three static methods for creating a TAIResult result. The TAIResult create methods take an int type as the first parameter. WebSphere Application Server expects the result to be a valid HTTP request return code and is interpreted in one of the following ways:

- If the value is `HttpServletResponse.SC_OK`, this response tells WebSphere Application Server that the TAI completed its negotiation. The response also tells WebSphere Application Server to use the information in the TAIResult result to create a user identity.
- Other values tell WebSphere Application Server to return the TAI output, which is placed into the HttpServletResponse response, to the web client. Typically, the web client provides additional information and then places another call to the TAI.

Table 106. TAIResults definitions. The created TAIResults results have the following meanings:

TAIResult	Explanation
public static TAIResult create(int status);	Indicates a status to WebSphere Application Server. The status cannot be SC_OK because the identity information is provided.
public static TAIResult create(int status, String principal);	Indicates a status to WebSphere Application Server and provides the user ID or the unique ID for this user. WebSphere Application Server creates credentials by querying the user registry.
public static TAIResult create(int status, String principal, Subject subject);	Indicates a status to WebSphere Application Server, the user ID or the unique ID for the user, and a custom Subject. If the Subject contains a hashtable, the principal is ignored. The contents of the Subject become part of the eventual user Subject.

All of the following examples are within the negotiateValidateandEstablishTrust method of a TAI.

The following code sample indicates that additional negotiation is required:

```
// Modify the HttpServletResponse object
//  The response code is meaningful only on the client
 return TAIResult.create(HttpServletResponse.SC_CONTINUE);
```

The following code sample indicates that the TAI determined the user identity. WebSphere Application Server receives the user ID only and queries the user registry for additional information:

```
// modify the HttpServletResponse object
 return TAIResult.create(HttpServletResponse.SC_OK, userid);
```

The following code sample indicates that the TAI determined the user identity. WebSphere Application Server receives the complete user information that is contained in the hashtable. In this code sample, the hashtable is placed in the public credential portion of the Subject:

```
// create Subject and place Hashtable in it
 Subject subject = new Subject;
 subject.getPublicCredentials().add(hashtable);
// the response code is meaningful for only the client
 return TAIResult.create(HttpServletResponse.SC_OK, "ignored", subject);
```

The following code sample indicates that an authentication failure occured. WebSphere Application Server fails the authentication request:

```
//log error message
// ....
 throw new WebTrustAssociationFailedException("TAI failed for this reason");
```

The following methods are additional methods on the TrustAssociationInterceptor interface. These methods are used for initialization, for shutdown, and for identifying the TAI to WebSphere Application Server. For more information, see the Java documentation.

Method name

public int initialize(Properties props)

Method result

This method is called during TAI initialization and is called only if custom properties are configured for the interceptor.

Method name

public String getVersion()

Method result

This method returns the version of the TAI.

Method name
public String getType()

Method result
This method returns the type of the TAI.

Method name
public void cleanup()

Method result
This method is called when stopping the WebSphere Application Server process. Stopping the WebSphere Application Server process provides an opportunity for the TAI to perform any necessary cleanup. This method is not necessary if cleanup is not required.

Enabling a plugpoint for custom password encryption

Two properties govern the protection of passwords. By configuring these two properties, you can enable a plugpoint for custom password encryption.

Before you begin

To view an example code sample that illustrates the com.ibm.wsspi.security.crypto.CustomPasswordEncryption interface, see “Plug point for custom password encryption” on page 985.

About this task

The encryption method is called for password processing whenever the custom class is configured and custom encryption is enabled. The decryption method is called whenever the custom class is configured and the password contains the {`custom:alias`} tag. The `custom:alias` tag is stripped prior to decryption.

Procedure

1. To enable custom password encryption, you must configure two properties:
   - **com.ibm.wsspi.security.crypto.customPasswordEncryptionClass** - Defines the custom class that implements the com.ibm.wsspi.security.crypto.CustomPasswordEncryption password encryption interface.
   - **com.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled** - Defines when the custom class is used for default password processing. When the passwordEncryptionEnabled option is not specified or set to `false`, and the passwordEncryptionClass class is specified, the decryption method is called whenever a {`custom:alias`} tag still exists in the configuration repository.
2. To configure custom password encryption, configure both of these properties in the `server.xml` file. How you perform this configuration is dependent on your existing directory structure. Choose one of the following ways to perform this configuration:
   - Place The custom encryption class (com.acme.myPasswordEncryptionClass) in a Java archive (JAR) file that resides in the `${WAS_INSTALL_ROOT}/classes` directory. In this case, you have created the `${WAS_INSTALL_ROOT}/classes` directory for this purpose.

     Note: WebSphere Application Server does not create the `${WAS_INSTALL_ROOT}/classes` directory.
   - Place the custom encryption class (com.acme.myPasswordEncryptionClass) in a Java archive (JAR) file that resides in the `${WAS_HOME}/lib/ext` directory or another valid existing directory.

   Every configuration document that contains a password (`security.xml` and any application bindings that contain RunAs passwords), must be saved before all of the passwords become encrypted with the custom encryption class.
3. If the custom implementation class defaults to the com.ibm.wsspi.security.crypto.CustomPasswordEncryptionImpl interface, and this class is present in the class path, then encryption is enabled by default. This simplifies the enablement process for all

nodes. It is not necessary to define any other properties except for those that the custom implementation requires. To disable encryption, but still use this class for decryption, specify the following class.

- com.ibm.wsspi.security.crypto.customPasswordEncryptionEnabled=false

What to do next

Whenever a custom encryption class encryption operation is called, and it creates a run-time exception or a defined PasswordEncryptException exception, the WebSphere Application Server runtime uses the {xor} algorithm to encode the password. This encoding prevents the storage of the password in plain text. After the problem with the custom class has been resolved, it automatically encrypts the password the next time the configuration document is saved.

When a RunAs role is assigned a user ID and password, it currently is encoded using the WebSphere Application Server encoding function. Therefore, after the custom plug point is configured to encrypt the passwords, it encrypts the passwords for the RunAs bindings as well. If the deployed application is moved to a cell that does not have the same encryption keys, or the custom encryption is not yet enabled, a login failure results because the password is not readable.

One of the responsibilities of the custom password encryption implementation is to manage the encryption keys. This class must decrypt any password that it encrypted. Any failure to decrypt a password renders that password to be unusable, and the password must be changed in the configuration. All encryption keys must be available for decryption there and no passwords are left using those keys. The master secret must be maintained by the custom password encryption class to protect the encryption keys.

You can manage the master secret by using a stash file for the keystore, or by using a password locator that enables the custom encryption class to locate the password so that it can be locked down.

Plug point for custom password encryption

A plug point for custom password encryption can be created to encrypt and decrypt all passwords in WebSphere Application Server that are currently encoded or decoded using Base64-encoding.

The implementation class of this plug point has the responsibility for managing keys, determining the encryption algorithm to use, and for protecting the master secret. The WebSphere Application Server runtime stores the encrypted passwords in their existing locations, preceded with {custom:alias} tags instead of {xor} tags. The custom part of the tag indicates that it is a custom algorithm. The alias part of the tag is specified by the custom implementation, which helps to indicate how the password is encrypted. The implementation can include the key alias, encryption algorithm, encryption mode, or encryption padding.

A custom provider of this plug point must implement an interface that is designed to encrypt and decrypt passwords. The interface is called by the WebSphere Application Server runtime whenever the custom plug point is enabled. The custom algorithm becomes one of the supported algorithms when the plug point is enabled. Other supported algorithms include {xor} (standard base64 encoding) and {os400} which is used on the iSeries platform.

The following example illustrates the com.ibm.wsspi.security.crypto.CustomPasswordEncryption interface:

```
package com.ibm.wsspi.security.crypto;
public interface CustomPasswordEncryption
{

    /**
     * The encrypt operation takes a UTF-8 encoded String in the form of a byte[].
     * The byte[] is generated from String.getBytes("UTF-8").
     * An encrypted byte[] is returned from the implementation in the EncryptedInfo
     * object.  Additionally, a logical key alias is returned in the EncryptedInfo
     * objectwhich is passed back into the decrypt method to determine which key was
     * used to encrypt this password. The WebSphere Application Server runtime has
     * no knowledge of the algorithm or the key used to encrypt the data.
     *
     * @param byte[]
```

```
     * @return com.ibm.wsspi.security.crypto.EncryptedInfo
     * @throws com.ibm.wsspi.security.crypto.PasswordEncryptException
     **/
    public EncryptedInfo encrypt (byte[] decrypted_bytes) throws PasswordEncryptException;

    /**
     * The decrypt operation takes the EncryptedInfo object containing a byte[]
     * and the logical key alias and converts it to the decrypted byte[].  The
     * WebSphere Application Server runtime converts the byte[] to a String
     * using new String (byte[], "UTF-8");
     *
     * @param com.ibm.wsspi.security.crypto.EncryptedInfo
     * @return byte[]
     * @throws com.ibm.wsspi.security.crypto.PasswordDecryptException
     **/
    public byte[] decrypt (EncryptedInfo info) throws PasswordDecryptException;

    /**
     * The following is reserved for future use and is currently not
     * called by the WebSphere Application Server runtime.
     *
     * @param java.util.HashMap
     **/
    public void initialize (java.util.HashMap initialization_data);
}
```

The com.ibm.wsspi.security.crypto.EncryptedInfo class contains the encrypted bytes with the user-defined alias that is associated with the encrypted bytes. This information is passed back into the encryption method to help determine how the password was originally encrypted.

```
package com.ibm.wsspi.security.crypto;
public class EncryptedInfo
{
    private byte[] bytes;
    private String alias;

/**
 * This constructor takes the encrypted bytes and a keyAlias as parameters.
 * This constructor is used to pass to or from the WebSphere Application Server
 * runtime to enable the runtime to associate the bytes with a specific key that
 * is used to encrypt the bytes.
 */

    public EncryptedInfo (byte[] encryptedBytes, String keyAlias)
    {
        bytes = encryptedBytes;
        alias = keyAlias;
    }

/**
 * This command returns the encrypted bytes.
 *
 * @return byte[]
 */
    public byte[] getEncryptedBytes()
    {
        return bytes;
    }

/**
 * This command returns the key alias.  The key alias is a logical string that is
 * associated with the encrypted password in the model. The format is
 * {custom:keyAlias}encrypted_password. Typically, just the key alias is placed
 * here, but algorithm information can also be returned.
 *
 * @return String
 */
    public String getKeyAlias()
    {
        return alias;
    }

}
```

The encryption method is called for password processing whenever the custom class is configured and custom encryption is enabled. The decryption method is called whenever the custom class is configured and the password contains the {`custom:alias`} tag . The `custom:alias` tag is stripped prior to decryption. For more information, see Enabling custom password encryption.

Implementing a custom authentication provider using JASPI

You can implement a custom authentication provider using Java Authentication SPI for Containers (JASPI, or sometimes called JASPIC) to handle the Java Platform, Enterprise Edition (Java EE) authentication of HTTP request and response messages destined for web applications.

Before you begin

For JASPI authentication processing to take place, application security must be enabled in the global or domain security configuration and the server must be restarted for the configuration changes to take effect. Read the Application security topic for more information.

About this task

This release of WebSphere Application Server supports the JSR 196: Java Authentication SPI for Containers (JASPI, or sometimes called JASPIC) specification, which enables third-party security providers to handle the Java Platform, Enterprise Edition (Java EE) authentication of HTTP request and response messages destined for web applications. The JASPI specification extends the pluggable authentication concepts of the Java Authentication and Authorization Service (JAAS) to the authentication of HTTP request and response messages. When application security is enabled, and a protected web resource is accessed, the web container and the security runtime collaborate to make an authentication decision for the caller. When using a third-party JASPI provider, the authentication decision is delegated to that provider.

The JASPI specification defines standard system programming interfaces that enable developers to write a pluggable custom authentication provider that can handle Java EE web authentication mechanisms as well as any extended authentication processing. The WebSphere Application Server runtime uses these standard system programming interfaces to invoke the JASPI authentication provider. Read the Servlet Container Profile section in the JSR 196: Java Authentication Service Provider Interface for Containers specification for the requirements that third-party authentication providers must satisfy for more information.

If application security is enabled with JASPI authentication, when the web resource (such as a servlet or a JavaServer Pages (JSP) file) is accessed, the security runtime checks if the web resource is mapped to a JASPI provider defined in the security configuration. If so, the runtime invokes the JASPI authentication provider to perform authentication for the HTTP request and response messages.

To implement a custom authentication provider using JASPI you must do the following:

Procedure

1. Develop a custom JASPI authentication provider.

   WebSphere Application Server provides support for the development of custom JASPI authentication providers to be used to perform authentication for the HTTP request and response messages destined for web applications. Read “Developing a custom JASPI authentication provider” on page 988 for more information.

2. Configure a new JASPI authentication provider.

   WebSphere Application Server allows an administrator to enable JASPI authentication and to define a third-party JASPI authentication provider as part of the global or domain security configuration. Read “Configuring a new JASPI authentication provider using the administrative console” on page 992 for more information.

3. Associate a JASPI authentication provider with an application or specific web modules.

   During application deployment, the administrator or deployer can use the Map JASPI Provider option to associate web applications and specific web modules with an existing JASPI authentication provider as defined in the security configuration. This association can also be made when editing the options for an existing installed application. By default, an application inherits the JASPI settings defined in the

WebSphere Application Server global or domain security configuration, and web modules inherit the application setting. The Map JASPI Provider option can be used to override these defaults. Read “Enabling JASPI authentication using the Map JASPI provider option during application deployment” on page 994 for more information.

Developing a custom JASPI authentication provider

You can develop a custom Java Authentication SPI for Containers (JASPI) authentication provider by creating classes that implement the required interfaces noted in the JSR 196: Java Authentication Service Provider Interface for Containers specification.

Before you begin

Review the specific interface implementation requirements for JASPI authentication providers and modules in the JSR 196: Java Authentication Service Provider Interface for Containers specification.

About this task

WebSphere Application Server supports the use of third-party authentication providers that are compliant with the servlet container profile specified in Java Authentication SPI for Containers (JASPI) Version 1.0.

The servlet container profile defines interfaces that are used by the security runtime environment in collaboration with the web container in WebSphere Application Server to invoke authentication modules before and after a web request is processed by an application. Authentication using JASPI modules is performed only when JASPI has been enabled in the security configuration and when a configured JASPI provider has been associated with the web module that processes the received web request.

To develop a custom authentication provider, create classes that implement the required interfaces noted in the JSR 196: Java Authentication Service Provider Interface for Containers specification. A provider can use one or more authentication modules for authentication. Modules can use callbacks to perform authentication, or they can manually add the necessary user identity information to the client subject. Depending on the scope of the provider, the implementation classes can be stored in various locations on the application server.

Procedure

1. Create a class that implements the javax.security.auth.message.config.AuthConfigProvider interface.

   The AuthConfigProvider implementation class must define a public two-argument constructor and the getServerAuthConfig public method:

   ```
   import java.util.Map;
   import javax.security.auth.callback.CallbackHandler;
   import javax.security.auth.message.AuthException;
   import javax.security.auth.message.config.AuthConfigFactory;
   import javax.security.auth.message.config.AuthConfigProvider;
   import javax.security.auth.message.config.ServerAuthConfig;

   public class SampleAuthConfigProvider implements AuthConfigProvider {

           public SampleAuthConfigProvider(Map<String, String> properties, AuthConfigFactory factory) {
                   ...
           }
           public ServerAuthConfig getServerAuthConfig(String layer, String appContext, CallbackHandler handler)
                   throws AuthException {
                   ...
           }
   }
   ```

   An instance of the AuthConfigProvider implementation class is used by WebSphere Application Server when a request arrives to be processed by the web module of the application. The getServerAuthConfig method is used to obtain a ServerAuthConfig instance. The CallbackHandler argument in the method call is used by the authentication module(s).
2. Create a class that implements the javax.security.auth.message.config.ServerAuthConfig interface.

The ServerAuthConfig implementation class must define the getAuthContextID and getAuthContext public methods:

```
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.message.AuthException;
import javax.security.auth.message.MessageInfo;
import javax.security.auth.message.config.ServerAuthConfig;
import javax.security.auth.message.config.ServerAuthContext;

public class SampleServerAuthConfig implements ServerAuthConfig {

        public String getAuthContextID(MessageInfo messageInfo) throws IllegalArgumentException {
                ...
        }
        public ServerAuthContext getAuthContext(String authContextID, Subject serviceSubject, Map properties)
                throws AuthException {
                ...
        }
}
```

The getAuthContextID and getAuthContext methods in the ServerAuthConfig implementation class are used to obtain a ServerAuthContext instance.

3. Create a class that implements the javax.security.auth.message.config.ServerAuthContext interface.

   The ServerAuthContext implementation class must define the validateRequest and secureResponse public methods:

```
import javax.security.auth.Subject;
import javax.security.auth.message.AuthException;
import javax.security.auth.message.AuthStatus;
import javax.security.auth.message.MessageInfo;
import javax.security.auth.message.config.ServerAuthContext;

public class SampleServerAuthContext implements ServerAuthContext {

        public AuthStatus validateRequest(MessageInfo messageInfo, Subject clientSubject, Subject serviceSubject)
                throws AuthException {
                ...
        }
        public AuthStatus secureResponse(MessageInfo messageInfo, Subject serviceSubject)
                throws AuthException {
                ...
        }
}
```

   The validateRequest method in the ServerAuthContext implementation class is used to invoke the module that authenticates the received web request message. If the authentication result is successful, the web container dispatches the received web request message that the target web module processes in the application. If the authentication result is not successful, the request is rejected with the appropriate response status.

4. Create a class that implements the javax.security.auth.message.module.ServerAuthModule interface.

   The ServerAuthModule implementation class must define the initialize, validateRequest, and secureResponse public methods:

```
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.message.AuthException;
import javax.security.auth.message.AuthStatus;
import javax.security.auth.message.MessageInfo;
import javax.security.auth.message.MessagePolicy;
import javax.security.auth.message.module.ServerAuthModule;

public class SampleAuthModule implements ServerAuthModule {

        public void initialize(MessagePolicy requestPolicy, MessagePolicy responsePolicy, CallbackHandler handler, Map options)
                throws AuthException {
                ...
        }

        public AuthStatus validateRequest(MessageInfo messageInfo, Subject clientSubject, Subject serviceSubject)
                throws AuthException {
                ...
        }
```

```
        public AuthStatus secureResponse(MessageInfo messageInfo, Subject serviceSubject)
                throws AuthException {
                ...
        }
}
```

The initialize method in the ServerAuthModule implementation class is called by the ServerAuthContext implementation class to initialize the authentication module and to associate it with the ServerAuthContext instance.

The validateRequest and secureResponse methods in this class are used respectively to authenticate the javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse contained in the javax.security.auth.message.MessageInfo that is received. These methods can use the CallbackHandler instance received in the initialize method to interact with the WebSphere security runtime to validate a user password, and the active user registry to retrieve a unique-id and membership groups for a user. The retrieved data is placed in a Hashtable in the set of private credentials in the client subject. The WebSphere Application Server implementation of CallbackHandler supports three callbacks:

- CallerPrincipalCallback
- GroupPrincipalCallback
- PasswordValidationCallback

WebSphere Application Server expects the name values obtained with PasswordValidationCallback.getUsername() and CallerPrincipalCallback.getName() to be identical. If they are not, unpredictable results occur. The CallbackHandler's handle() method processes each callback given in the argument array of the method sequentially. Therefore, the name value set in the private credentials of the client subject is the one obtained from the last callback processed.

Note: Always use PasswordValidationCallback to validate a user password and to add the appropriate credentials to the client subject during authentication:

```
        import javax.security.auth.Subject;
        import javax.security.auth.callback.Callback;
        import javax.security.auth.message.AuthException;
        import javax.security.auth.message.AuthStatus;
        import javax.security.auth.message.MessageInfo;
        import javax.security.auth.message.callback.PasswordValidationCallback;

        public AuthStatus validateRequest(MessageInfo messageInfo, Subject clientSubject, Subject serviceSubject)
                throws AuthException {
                ...
                PasswordValidationCallback pvcb = new PasswordValidationCallback(clientSubject, username, password);
                handler.handle(new Callback[] {pvcb});
                ...
        }
```

If CallbackHandler is not used by the authentication module, and validateRequest returns a successful status, WebSphere Application Server requires that a Hashtable instance be included in the clientSubject with user identity information so that a custom login can be performed to obtain the credentials for the user. This Hashtable can be added to the client subject as in the following example:

```
import java.util.Hashtable;
import java.util.String;
import javax.security.auth.Subject;
import javax.security.auth.message.AuthException;
import javax.security.auth.message.AuthStatus;
import javax.security.auth.message.MessageInfo;
import com.ibm.wsspi.security.registry.RegistryHelper;
import com.ibm.wsspi.security.token.AttributeNameConstants.AttributeNameConstants;

public AuthStatus validateRequest(MessageInfo messageInfo, Subject clientSubject, Subject serviceSubject)
        throws AuthException {
        ...
        UserRegistry reg = RegistryHelper.getUserRegistry(null);
        String uniqueid = reg.getUniqueUserID(username);

        Hashtable hashtable = new Hashtable();
        hashtable.put(AttributeNameConstants.WSCREDENTIAL_UNIQUEID, uniqueid);
        hashtable.put(AttributeNameConstants.WSCREDENTIAL_SECURITYNAME, username);
        hashtable.put(AttributeNameConstants.WSCREDENTIAL_PASSWORD, password);
```

```
        hashtable.put(AttributeNameConstants.WSCREDENTIAL_GROUPS, groupList); //optional
        clientSubject.getPrivateCredentials().add(hashtable);
        ...
}
```

For more information about the Hashtable requirements and custom login, read about Developing custom login modules for a system login configuration for JAAS.

To support the login and authenticate methods of the Java Servlet 3.0 specification, the following logic must be added to the validateRequest method in the ServerAuthModule implementation class:

```
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.message.AuthException;
import javax.security.auth.message.AuthStatus;
import javax.security.auth.message.MessageInfo;
import javax.servlet.http.HttpServletRequest;

public AuthStatus validateRequest(MessageInfo messageInfo, Subject clientSubject, Subject serviceSubject)
        throws AuthException {
        ...
        Map msgMap = messageInfo.getMap();

        if ("login".equalsIgnoreCase(msgMap.get("com.ibm.websphere.jaspi.request"))) {
                // This request is for the login method
                String username = msgMap.get("com.ibm.websphere.jaspi.user");
                String password = msgMap.get("com.ibm.websphere.jaspi.password");
                // Authenticate using the user name and password set above.
        }
        else if ("authenticate".equalsIgnoreCase(msgMap.get("com.ibm.websphere.jaspi.request"))) {
                // this request is for the authenticate method
                String authHeader
                     = ((HttpServletRequest) messageInfo.getRequestMessage()).getHeader("Authorization");
                if (authHeader == null) {
                     // The user has not provided a username and password yet, return
                     // AuthStatus.SEND_CONTINUE to challenge
                }
                else {
                     // Authenticate using the user name and password in the authentication header.
                }
        }
        else {
                // This is not a Servlet 3.0 login or authenticate request; handle as usual.
        }
        ...
}
```

5. Compile all newly created classes.

   The following JAR files in your WebSphere Application Server installation must be specified in the class path to successfully compile the new classes:

   - *app_server_root*/dev/JavaEE/j2ee.jar
   - *app_server_root*/dev/was_public.jar (if any public WebSphere APIs were used)

6. Create a JAR file with the compiled classes.

   Depending on the requirements, the JAR file can be placed in one of three locations:

   - *app_server_root*/lib

     This location is always on the classpath for the WebSphere Application Server classloader. Using this location, the provider can be registered for a set of web modules or applications as the cell or domain default provider for all web modules and applications, and it can be registered manually as a persistent provider.

   - Shared library

     Place the provider JAR file anywhere on the WebSphere Application Server system. Configure a shared library that points to the JAR, and add that shared library to the application or server classpath. In a shared library, the provider can be registered for a set of web modules or applications, but the provider cannot be used as the cell or domain default provider. It also cannot

be registered as a persistent provider because the shared library is not in the classpath for provider registration during server startup. For more information about configuring a shared library, read about Creating shared libraries.

- Embedded in the application

  Include the provider JAR file in the application's EAR file as a utility JAR, or embed the compiled class files in the web module WAR. The embedded provider can be registered for the web modules in the application as long as the classes are included in the classpath for the web module. This provider cannot be used as a cell or domain default provider, nor can it be registered as a persistent provider. The classes in the application are not available for provider registration during server startup.

7. Configure the provider in the security configuration using the administrative console or an administration script.

   Read about “Configuring a new JASPI authentication provider using the administrative console” for more information.

Configuring a new JASPI authentication provider using the administrative console

You can configure a new Java Authentication SPI (JASPI) authentication provider in the cell or in the given security domain by using the administrative console.

About this task

This release of WebSphere Application Server supports integration of message authentication providers that are compliant with the JASPI for Containers Version 1.0 specification.

When JASPI authentication providers are configured, and WebSphere Application Server receives an HTTP request message, the security runtime environment determines if the target application is configured to use JASPI authentication. If so, the runtime environment invokes the selected authentication provider to validate the received message. Otherwise, authentication of the message request is done according to the authentication mechanism provided by WebSphere Application Server for the appropriate messaging layer.

If you want to use JASPI message authentication services, you must supply an implementation of the required interfaces as defined in the JASPI specification. Read “Developing a custom JASPI authentication provider” on page 988 for more information on these interfaces.

Authentication of HTTP request and response messages destined for JASPI-enabled deployed applications is performed according to the requirements of the Servlet Container Profile specified in the new specification.

Note: JASPI is supported in a mixed-cell environment, but can only be used in nodes that are version 8 or higher. Back-level nodes use existing authentication mechanisms.

To configure a new JASPI authentication provider using the administrative console, do the following:

Procedure

1. Click **Security > Global security**.
2. Select **Enable Java Authentication SPI (JASPI)** to enable support for JASPI authentication.
3. Click **Providers**.

   Note: It is not necessary to select **Enable Java Authentication SPI (JASPI)** until after you have configured a new JASPI authentication provider.

   Note: The Default provider option is used to specify a single JASPI authentication provider to perform authentication for all web modules when JASPI authentication is enabled, and you do not override the web module to JASPI provider mapping during application deployment. During

application deployment, you can override the default for every web module where it does not apply by choosing not to use JASPI or by naming a different provider to use for authentication. However, it is not recommended that you use this option unless you are certain that your default provider is capable of handling all types of web authentication (basic authentication, form authentication and client certificate authentication).

4. Click **New**.
5. Enter a name that uniquely identifies the JASPI authentication provider in the Provider name field.
6. Optional: Enter a textual description of the authentication provider in the Description field.
7. Enter the package-qualified name of the class that implements the authentication provider interface (javax.security.auth.message.config.AuthConfigProvider) in the Class name field.

   Note: In the Message layer field, WebSphere Application Server Version 8.5 supports only the HttpServlet message layer profile as defined in the JASPI specification. You cannot change this value.
8. Optional: Under Custom Properties, click **New** if you require more than one property. This parameter is a list of key/value pairs.
9. Click **OK** or **Apply**.

What to do next

You can also configure a new JASPI authentication provider by using **wsadmin** commands. Read JaspiManagement command group for the AdminTask object for more information.

Verify that your server has been restarted so that the changes to configure the JASPI provider will take effect.

Modifying an existing JASPI authentication provider using the administrative console

You can modify and configure an existing Java Authentication SPI (JASPI) authentication provider in the cell or in the given security domain by using the administrative console.

About this task

To modify and configure an existing JASPI authentication provider using the administrative console, do the following:

Procedure

1. Click **Security > Global security**.
2. Click **Providers**. You also have the option to change the Default provider from the drop-down list.

   Note: You can modify the value of the **Enable Java Authentication SPI (JASPI)** checkbox to indicate whether or not JASPI support is enabled at a later time.
3. Select an existing JASPI authentication provider to modify.
4. Optional: Enter a textual description of the authentication provider in the Description field.
5. Enter a new package-qualified name of the class that implements the authentication provider interface (javax.security.auth.message.config.AuthConfigProvider) in the Class name field if you wish to change it.

   Note: In the Message layer field, WebSphere Application Server Version 8.5 supports only the HttpServlet message layer profile as defined in the JASPI specification. You cannot change this value.

6. Optional: Under Custom Properties, select an existing custom configuration property. Click **Delete** to remove the property, **Edit** to modify the property, or click **New** to create a new property. If you select **Edit** to modify an existing property, you can enter new values for the Name field and Value field if you wish to change them.
7. Click **OK** or **Apply**.

What to do next

You can also modify an existing JASPI authentication provider by using **wsadmin** commands. For more information, read JaspiManagement command group for the AdminTask object.

Verify that your server has been restarted so that the changes to configure the JASPI provider will take effect.

Deleting a JASPI authentication provider using the administrative console

You can delete an existing Java Authentication SPI (JASPI) authentication provider in the cell or in the given security domain by using the administrative console.

About this task

To delete an existing JASPI authentication provider using the administrative console, do the following:

Procedure

1. Click **Security** > **Global security**.
2. Click **Providers**, You can optionally select or deselect the **Enable Java Authentication SPI (JASPI)** check box.

   Note: You can modify the value of the **Enable Java Authentication SPI (JASPI)** checkbox to indicate whether or not JASPI support is enabled at a later time.
3. Select an existing JASPI authentication provider to delete.
4. Click **Delete**.

What to do next

You can also delete a JASPI authentication provider by using wsadmin commands. For more information, read JaspiManagement command group for the AdminTask object.

Verify that your server has been restarted so that the changes to the JASPI provider configuration will take effect.

Enabling JASPI authentication using the Map JASPI provider option during application deployment

An administrator or deployer can use the Map JASPI Provider option during application deployment to associate web applications and specific web modules with an existing Java Authentication SPI (JASPI) authentication provider as defined in the security configuration. This association can also be made when editing the options for a previously installed application.

Before you begin

Before you perform this task, verify that a JASPI authentication provider is defined as part of the global or domain security configuration. Read about “Configuring a new JASPI authentication provider using the administrative console” on page 992 for more information.

About this task

By default, an application inherits the JASPI settings defined in the WebSphere Application Server global or domain security configuration, and web modules inherit the application setting. However, you can override these default values by using the Map JASPI Provider option during application deployment. Use this option to associate a specific JASPI provider from the global or domain security configuration with the entire application or with specific web modules. You can also use this option to specify that JASPI authentication not be used for an application or specific web module.

To associate a web application or specific web modules with an existing JASPI provider:

Procedure

1. From the administrative console, click **Applications > New Application > New Enterprise Application**. Complete the required steps until you see the step for Map JASPI Provider, or click the **Map JASPI Provider** step from the installation options. A list containing the application name and associated web modules is displayed. To update a JASPI provider association after an application has been deployed, click **Applications > Application Types > WebSphere enterprise applications**, and then select the application to be modified. Click **JASPI Provider** under the Detail properties.
2. Select the application or specific web module for which the JASPI provider setting is to be modified.
3. Click the **Select JASPI Provider** menu and select one of the following options:

   Do not use JASPI
   Select to disable JASPI authentication for the selected web module or for the application.

   Inherit JASPI provider
   Select to inherit the JASPI authentication settings from default values in the cell or domain security configuration, as appropriate.

   When **Inherit JASPI provider** is selected for a web module, JASPI authentication settings for the selected module are the settings that are specified for the application.

   When **Inherit JASPI provider** is selected for the application, JASPI authentication settings are the settings that are specified in the appropriate cell or domain security configuration.

   Provider name
   When a specific provider name is selected, that provider name is used to perform authentication of web requests for the selected application or web module.
4. Complete the remaining steps to finish installing and deploying the application.

What to do next

Verify that your server has been restarted to ensure that the configuration changes to define the JASPI provider take effect. Read about “Configuring a new JASPI authentication provider using the administrative console” on page 992 for more information.

JASPI authentication providers collection

The Java Authentication Service Provider Interface (JASPI) for Containers Version 1.0 specification defines standard system programming interfaces that enable developers to write a pluggable custom authentication provider that can handle Java EE web authentication mechanisms as well as any extended authentication processing. The WebSphere Application Server runtime uses these standard system programming interfaces to invoke the JASPI authentication provider.

Read the Servlet Container Profile section in the JSR 196: Java Authentication Service Provider Interface for Containers specification for the requirements that third-party authentication providers must satisfy for more information.

If application security is enabled, and JASPI authentication is enabled with providers configured, when a web resource (such as a servlet or a JavaServer Page (JSP) file) is accessed, the security runtime checks

if the web resource is mapped to a JASPI provider defined in the security configuration. If so, the runtime invokes the JASPI authentication provider to perform authentication for the HTTP request and response messages.

Note: WebSphere Application Server Version 8.5 supports only the HttpServlet message layer profile as defined in the JASPI specification.

To view this administrative console page, click **Security > Global security**. Under Authentication, click **Providers**.

To configure a new custom JASPI authentication provider in the cell or in the given security domain, click **New** and specify provider settings.

Provider name:

Specifies a name that uniquely identifies the authentication provider.

Select an existing custom JASPI authentication provider name to edit and configure it.

JASPI authentication provider details

Use this page to provide configuration details for your custom Java Authentication SPI (JASPI) authentication service provider.

To view this administrative console page, click **Security > Global security**. Under Authentication, click **Providers**. Select an existing authentication service provider name or click **New** to create a new one.

Provider name:

Specifies a name that uniquely identifies the authentication provider.

Description:

Specifies a textual description of the authentication provider.

Class name:

Specifies the package-qualified name of the class that implements the authentication provider interface (javax.security.auth.message.config.AuthConfigProvider).

Message layer:

WebSphere Application Server Version 8.5 supports only the HttpServlet message layer profile as defined in the JASPI specification.

Custom properties:

Specifies additional custom properties needed to initialize the authentication provider. This parameter is a list of key/value pairs.

Click **Delete** to remove a custom property or **Edit** to modify a custom property.

JASPI authentication enablement for applications

Use this page to enable or disable Java Authentication SPI (JASPI) authentication for an application or web module, and to specify the name of a JASPI authentication provider to be used for authenticating messages for the application or web module.

To view this administrative console page, click **Applications > Application Types > WebSphere enterprise applications**. Select an application, and under Detail Properties, select **JASPI provider**.

Select JASPI provider:

Select to indicate the web modules in the application that you wish to specify or to override the default JASPI authentication settings for.

Select one of the JASPI provider names to choose a provider to use to perform authentication of web requests for the selected Web module or the application.

To specify how JASPI authentication is performed for the selected web module or the application, choose one of the following:

Do not use JASPI
Select to disable JASPI authentication for the selected web module or for the application.

Inherit JASPI provider
Select to inherit the JASPI authentication settings from default values in the cell or domain security configuration, as appropriate.

When Inherit JASPI provider is selected for a web module, JASPI authentication settings for the selected module are the settings that are specified for the application.

When Inherit JASPI provider is selected for the application, JASPI authentication settings are the settings that are specified in the appropriate cell or domain security configuration.

Provider name
When a specific provider name is selected, that provider is used to perform authentication of web requests for the selected application or web module.

If JASPI authentication is enabled, and a specific provider name is not specified, then the default provider name is used. For more information, read about configuring a new JASPI authentication provider using the administrative console.

If JASPI authentication is disabled, or if no default provider is selected, JASPI authentication is not performed. Web authentication is then performed according to another authentication mechanism as selected in the cell or domain security configuration.

Chapter 22. Developing Startup beans

This page provides a starting point for finding information about startup beans.

Startup beans allow business logic to run when an application starts or stops.

Using startup beans

There are two types of startup beans: application startup beans and Module startup beans.

About this task

Note: The capabilities provided with startup singleton session beans (EJB 3.1 specification) causes the WebSphere Application Server proprietary startup beans function to be deprecated.

A module startup bean is a session bean that is loaded when an EJB Jar file starts. Module startup beans enable Java Platform Enterprise Edition (Java EE) applications to run business logic automatically, whenever an EJB module starts or stops normally. An application startup bean is a session bean that is loaded when an application starts. Application startup beans enable Java EE applications to run business logic automatically, whenever an application starts or stops normally.

Startup beans are especially useful when used with asynchronous bean features. For example, a startup bean might create an alarm object that uses the Java Message Service (JMS) to periodically publish heartbeat messages on a well-known topic. This enables clients or other server applications to determine whether the application is available. Refer to the Enabling an application to wait for a messaging engine to start article if you are using the default JMS provider.

Procedure

1. For Application startup beans, use the home interface, com.ibm.websphere.startupservice.AppStartUpHome, to designate a bean as an Application startup bean. For Module startup beans, use the home interface, com.ibm.websphere.startupservice.ModStartUpHome, to designate a bean as a Module startup bean.
2. For Application startup beans, use the remote interface, com.ibm.websphere.startupservice.AppStartUp, to define start() and stop() methods on the bean. For Module startup beans, use the remote interface, com.ibm.websphere.startupservice.ModStartUp, to define start() and stop() methods on the bean.

   The startup bean start() method is called when the module or application starts and contains business logic to be run at module or application start time.

   The start() method returns a boolean value. **True** indicates that the business logic within the start() method ran successfully. Conversely, **False** indicates that the business logic within the start() method failed to run completely. A return value of False also indicates to the Application server that application startup is aborted.

   The startup bean stop() methods are called when the module or application stops and contains business logic to be run at module or application stop time. Any exception thrown by a stop() method is logged only. No other action is taken.

   The start() and stop() methods must never use the TX_MANDATORY transaction attribute. A global transaction does not exist on the thread when the start() or stop() methods are invoked. Any other TX_* attribute can be used. If TX_MANDATORY is used, an exception is logged, and the application start is aborted.

   The start() and stop() methods on the remote interface use **Run-As** mode. **Run-As** mode specifies the credential information to be used by the security service to determine the permissions that a principal has on various resources. If security is on, the **Run-As** mode needs to be defined on all of the methods called. The identity of the bean without this setting is undefined.

There are no restrictions on what code the start() and stop() methods can run, since the full Application Server programming model is available to these methods.

3. Use an *optional* environment property integer, `wasStartupPriority`, to specify the start order of multiple startup beans in the same Java Archive (JAR) file. If the environment property is found and is the wrong type, application startup is aborted. If no priority value is specified, a default priority of `0` is used. It is recommended that you specify the priority property. Beans that have specified a priority are sorted using this property. Beans with numerically lower priorities are run first. Beans that have the same priority are run in an undefined order. All priorities must be positive integers. Beans are stopped in the opposite order to their start priority. The priority values for module startup beans and application startup beans are mutually exclusive. All modules will be started prior to the application being declared as "started" and therefore the start() methods for module startup beans within an application will be invoked prior to the start() methods for any application startup beans. Likewise, all application startup bean stop() methods for a specific Java Archive (JAR) file will be invoked prior to any module startup bean stop() methods for that JAR.

   Note: The `wasStartupPriority` environment property integer cannot be set through either a command or the administrative console. This environment property integer is an EJB environment entry that is to be set by an application developer not an administrator. You set the integer value in the `ejb-jar.xml` file as shown in the example below:

   ```
   <env-entry>
     <env-entry-name>wasStartupPriority</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>3</env-entry-value>
   </env-entry>
   ```

   As with any other EJB environment entry, you set a separate `wasStartupPriority` value for each EJB.

4. For module startup beans, the order in which EJB modules are started can be adjusted via the "Starting weight" value associated with each module
5. To control who can invoke startup bean methods via WebSphere Security do the following:
   a. Define the method permissions for the Start() and Stop() methods as you would for any EJB module. (See "Defining method permissions for EJB modules".)
   b. Ensure that the user that is mapped to the Security Role defined for the startup bean methods is the same user that is defined as the Server user ID within the User Registry.

What to do next

View the startup beans service settings.

Enabling startup beans in the administrative console

Enabling startup beans in the administrative console enables Java 2 Platform Enterprise Edition (J2EE) applications to run business logic automatically, whenever an application starts or stops normally.

About this task

Use the following steps to enable startup beans in the administrative console.

Procedure

1. Start the administrative console.
2. Select **Servers** > **Application Servers** > *server_name* > **Container Services** > **Startup beans service**.
3. Select the **Enable service at server startup** check box.
4. Click **Apply** to save the configuration.

What to do next

View the startup beans service settings.

Startup beans service settings

Use this page to enable startup beans that control whether application-defined startup beans function on this server. Startup beans are session beans that run business logic through the invocation of start and stop methods when applications start and stop. If the startup beans service is disabled, then the automatic invocation of the start and stop methods does not occur for deployed startup beans when the parent application starts or stops. This service is disabled by default. Enable this service only when you want to use startup beans. Startup beans are especially useful when used with asynchronous beans.

Note: The capabilities provided with startup singleton session beans (EJB 3.1 specification) causes the WebSphere Application Server proprietary startup beans function to be deprecated.

To view this administrative console page, click **Servers > Server types > WebSphere application servers >** *server_name*. Under **Container Settings**, expand **Container Services** then click **Startup beans service**.

Enable service at server startup

Specifies whether the server attempts to initiate the startup beans service.

Information	**Value**
Default	Cleared
Range	**Selected** When the application server starts, it attempts to initiate the startup bean service automatically. **Cleared** The server does not try to initiate the startup beans service. All startup beans do not start or stop with the application. If you use startup beans on this server, then the system administrator must start the startup beans service manually or select this property, and then restart the server.

Chapter 23. Developing Service integration

This page provides a starting point for finding information about service integration.

Service integration provides asynchronous messaging services. In asynchronous messaging, producing applications do not send messages directly to consuming applications. Instead, they send messages to destinations. Consuming applications receive messages from these destinations. A producing application can send a message and then continue processing without waiting until a consuming application receives the message. If necessary, the destination stores the message until the consuming application is ready to receive it.

Programming mediations

Several distinct tasks are involved in programming a mediation. Typically, the mediation code is written by a programmer, and is then deployed and administered by an integrator.

Before you begin

Code examples for writing a mediation are provided in the following topic: “Adding mediation function to handler code” on page 1005.

The following application programming interfaces are provided for you to work with messages:

- You can use the SIMessage API to manipulate the contents of the message.
- You can use the SIMediationSession API to access the service integration bus so that your mediation can send and retrieve messages.

You can deploy mediations by using tooling such as the Rational Application Developer tools.

About this task

The tasks for programming a mediation are as follows:

Developing
Writing a mediation by adding functional code to a mediation handler.

Deploying
Adding a mediation to a mediation handler list, and deploying it.

Administering
Associating a mediation handler with a destination (optional), and configuring the parameters to be used by the mediation handler at run time.

Take the following steps to program a mediation:

Procedure

1. Create a mediation handler. For more information, see “Writing a mediation handler” on page 1004.
2. Add mediation function code to your mediation handler. For more information, see “Adding mediation function to handler code” on page 1005.
3. Optional: Working with the message payload, for example for logging messages within a mediation. For more information, see “Working with the message payload” on page 1012.
4. Use the Rational Application Developer tools to create a handler list, add your mediation handler to the list, and deploy the handler list as an Enterprise Archive (EAR file). See the Rational Application Developer information center for information about how to do this.

Serializing the content of SIMessage

Use this task to convert an SIMessage object to a byte array.

About this task

If you want to save an SIMessage object in your local file system or in a database, you must first convert the object to a byte array and format string. You can reconstruct the message from the byte array and format string. To do this, complete the following steps.

Procedure

1. In your application program, record the format string associated with the SIMessage instance. For example:

   ```
   String savedFormat=message.getFormat();
   ```

2. Call the `getDataGraphAsBytes`. For example:

   ```
   Bytes newDataGraph = message.getNewDataGraph(newFormat);
   ```

   This method returns a copy of the payload as a byte stream. You can store the bytes and the associated format string, as you require.

3. Optional: To reconstruct the message, call the method `createDataGraph` provided by the SIDataGraphFactory API. This method requires a byte array and a format string. For example:

```
DataGraph newDataGraph = SIDataGraphFactory.getInstance().createDataGraph(byteArray, newFormat);
```

   This method creates a new data graph by parsing the bytes according to the format passed to the method.

What to do next

You can use the newly created datagraph as the payload of an SIMessage instance by using the SIMessage setDataGraph() method. For example:

```
newMessage.setDataGraph(newDataGraph, savedFormat);
```

Writing a mediation handler

You can write a mediation handler, add mediation function to it, and prepare it for installation on an application server.

Before you begin

You should have access to a Java programming environment, and an assembly tool such as IBM Rational Application Developer.

About this task

A mediation handler can be deployed. Each mediation handler executes some specific message processing at run time, for example transforming a message format, or routing a message to a particular destination. A mediation handler is a Java program framework, to which you add the code that performs the mediation function.

You can define a mediation handler class either in a Java project or an EJB project (which is needed for the deployment artifact). Your programming and deployment artifacts can be separated in different projects. The steps below are for an EJB project, but the steps are very similar if you want to create a Java project, because you define a target server for either a Java project or an EJB project and the server runtime plug-in sets the class path correctly.

Procedure

1. Create a new EJB project:
   a. Switch to the Java EE perspective to work with Java EE projects. Click **Window > Open Perspective > Other > Java EE**.
   b. From the File menu, select **New > Project**.
   c. Expand the Java EE folder, and select Enterprise Application Project. Click **Next**.
   d. Optional: If you have created a Java project instead of an EJB project, right click on the Java project folder icon for the context menu and select Properties. When the Properties panel appears, select the Server properties and target the project to an appropriate server for your system, as in the next step.
   e. Enter a name for the project and target the project to an appropriate server for your system. (If this is the first time you target this server, click **New...**.) Click **Next** to take you to the EAR Module Projects window.
   f. Click **New Module...**.
   g. Create a new module project by selecting the check box against EJB project, and entering the name of your mediation handler.
   h. Click **Finish**. You are returned to the EAR Module Projects window.
   i. Click **Finish** to create the new project.
2. Create a mediation handler class by implementing the `com.ibm.websphere.sib.mediation.handler.MediationHandler` interface.
   a. From the File menu, select **New > Java Class**.
   b. Specify the source folder for your mediation EAR project.
   c. Specify a name for your mediation handler.
   d. Select Superclass `java.lang.Object`.
   e. Select Interface `com.ibm.websphere.sib.mediation.handler.MediationHandler`.
   f. Select the **Inherited abstract methods** check box.
   g. Click **Finish** to create the new mediation handler class.
3. Add functional code that transforms or routes messages to your mediation handler by using the IBM Rational Application Developer. For more information, see “Adding mediation function to handler code.” Beware that the default return value for the handle method created by the toolkit is `false`, which causes the message to be discarded. You must change the return value to `true` to preserve the message.
4. Generate an EAR file from your mediation handler class. Follow the instructions in the IBM Rational Application Developer documentation.

What to do next

Next, you are ready to install the EAR file containing your mediation handler into the application server.

Adding mediation function to handler code

A mediation handler is a wrapper for the mediation code that operates on the message. You can add mediation function to an existing mediation handler by working with the message context, the message properties, the message header or the message payload.

Before you begin

Create or open a mediation handler in an EJB project. For more information, see “Writing a mediation handler” on page 1004.

About this task

There are four main aspects of the message through which you can change the behavior of the mediation handler code, or influence the routing of a message. You can change the values in the message context and message properties, and you can work with the contents of the message (called the message payload), or with the message header:

- “Working with the message context”
- “Working with message properties” on page 1007
- “Working with the message header” on page 1008
- “Working with the message payload” on page 1012

Example: Using mediations to trace, monitor and log messages

The most straightforward use of a mediation is for tracing, monitoring or logging messages that pass through a destination or topics spaces. This type of mediation does not modify the message; it only extracts information from the message and saves or displays the information elsewhere.

For example, the following mediation handler displays the API message and correlation IDs for each message it is sent:

```
public boolean(MessageContext context)
{
  SIMessageContext   msgCtx  = (SIMessageContext)context;
  SIMediationSession session = msgCtx.getSession();
  SIMessage          msg     = msgCtx.getMessage();
  String             msgId   = msg.getApiMessageId();
  String             corrId  = msg.getCorrelationId();
  String             dest    = session.getDestinationName();

  System.out.println(msgId+" (correlation id="+corrid) is passing through "+dest+".");

  return true;
}
```

Working with the message context

You can work with the message properties to affect the way a message is mediated.

Before you begin

Before you start this task, you should read about how information is carried in the mediation context in Mediation context information

About this task

Interface SIMessageContext has a superinterface MessageContext. Methods in MessageContext allow you to manage a set of message properties, which enable handlers in a handler chain to share processing-related state. Most importantly, you can get the value of a specific property from the MessageContext by using the method getProperty, and you can set the name and value of a property associated with the MessageContext by using the method setProperty. You can also view the names of the properties in this MessageContext and remove a property (that is, a name-value pair) from the MessageContext.

At mediation runtime, all of the user-defined properties that have been set during configuration for the current mediation (see Configuring mediation context properties) are applied to the MediationContext property set.

Procedure

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. As you are working with the MessageContext methods that give you access to message properties, you do not have to cast the interface to SIMessageContext **unless** you are also interested in the methods provided by SIMessageContext.
2. Get the SIMessage from the MessageContext object. For example, `SIMessage message = ((SIMessageContext)context).getSIMessage();`
3. Retrieve or set properties, by using the MessageContext methods. For instance, if a property has been defined during configuration with the name streetName, the type String, and the value "Main Street" your code to retrieve and print the street name might look like this:

Example

```
public boolean handle(MessageContext context) throws MessageContextException {

   ........
  {
    /* Retrieve the street name property */
    String myStreetName;
    myStreetName = (String) getProperty(streetName);

    /* Display property value */
    System.out.println(myStreetName);

  }
}
```

Working with message properties

You can work with the message properties to affect subsequent processing.

Before you begin

Before you start this task, you should read about the properties that are supported by the SIMessage interface in Message properties support for mediations.

About this task

There are two different types of message properties:

- System properties (including JMS headers, JMSX properties, and JMS_IBM_properties)
- User properties.

You can work with message properties to affect which messages a later mediation should process, or to affect processing by a downstream application or mediation. The rule set in the selector field during mediation configuration tests values in the message properties.

You can access, modify and clear properties by using the SIMessage interface (see “SIMessage” on page 1016.) There are three different sets of methods:

- These properties operate on system properties, plus user properties if the name is qualified with a prefix `user.`:
  - getMessageProperty
  - setMessageProperty
  - deleteMessageProperty
  - clearMessageProperties
- These properties operate on user properties only, without the need for the prefix `user.`:
  - getUserProperty

  - setUserProperty
  - deletUserProperty
  - clearUserProperties
- getUserPropertyNames returns a list of the names of the user properties in the message.

Typically, you can work with message properties in the following way, when programming a mediation:

Procedure

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext **unless** you are only interested in the methods provided by MessageContext.
2. Get the SIMessage from the MessageContext object. For example, `SIMessage message = ((SIMessageContext)context).getSIMessage();`
3. Build your mediation header function in a similar way to these examples, by using the reference information in Message properties support for mediations to help:
   a. Get a user property of the message. For instance, `String task = (String)msg1.getUserProperty("task");`. In this case, the task string might refer to an operation that the mediation should perform.
   b. Set a user property, where message Properties are stored as name-value pairs. The setUserProperty method might only be used to set user properties, so the name passed into the method should not include the "user." prefix. For example, `msg1.setUserProperty("background","green");`
   c. Delete a user property from the message. For instance, `msg1.deleteUserProperty("task");`

Example

Mediation function code to work with message properties might look similar to the code snippet in this example:

```
String task = (String)msg1.getUserProperty("task");
if (task != null) {
  if (task.equals("addColor")) {
    msg1.setMessageProperty(SIProperties.JMS_IBM_Format, "colorful");
    msg1.setUserProperty("background","green");
    msg1.setUserProperty("foreground","purple");
    msg1.setUserProperty("depth",new Integer(3));
    msg1.deleteUserProperty("task");
  }
  else {
    msg1.clearUserProperties();
  }
}
```

Working with the message header

You can add function to a preexisting mediation handler to operate on the message header.

Before you begin

Create or open a basic mediation handler in an EJB project (see “Writing a mediation handler” on page 1004. It is useful to have understood the elements of the task “Working with the message payload” on page 1012, because some of those elements are used in this task

About this task

There are different types of field that you can set in message headers. Importantly, you can set the forward and return routing addresses for messages after they have been mediated at the current

destination. In addition there are other fields that you can set, such as priority and reliability for the message and its reply, and the remaining time before the message (or the reply) expires.

Procedure

1. To set routing addresses in the message header, see “Setting routing addresses in a message header.”
2. To set all other fields in the message header, see “Working with non-routing path fields in a message header” on page 1011.

Setting routing addresses in a message header:

You can add function to a pre-existing mediation handler to set routing addresses in the message header.

Before you begin

Before you start this task, you should have created the basic mediation handler in an EJB project (see “Writing a mediation handler” on page 1004.

About this task

To work with routing addresses, you will use the SIDestinationAddress and SIDestinationAddressFactory APIs. The SIDestinationAddress is the public interface that represents an service integration bus, and gives your mediation access to the name of the destination and the bus name. SIDestinationAddressFactory enables you to create a new SIDestinationAddress to represent an service integration bus destination. For reference information about these APIs, see “SIDestinationAddress” on page 1010 and “SIDestinationAddressFactory” on page 1011.

Procedure

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext **unless** you are only interested in the methods provided by MessageContext.
2. Get the SIMessage from the MessageContext object. For example:
   ```
   SIMessage message = ((SIMessageContext)context).getSIMessage();
   ```
3. Build your mediation header function by using these basic steps:
   a. Get a handle to the runtime environment. For example:
      ```
      .... SIMediationSession mediationSession = mediationContext.getSession();
      ```
   b. Create a forward routing path to set on the cloned object. For example, use the Vector class to create a extendable array of objects.
   c. Get the SIDestinationAddressFactory that is to be used for creating SIDestinationAddress instances. For example:
      ```
      SIDestinationAddressFactory destFactory = SIDestinationAddressFactory.getInstance();
      ```
   d. Create a new SIDestinationAddress, representing a service integration bus destination. For example:
      ```
      SIDestinationAddress dest = destFactory.createSIDestinationAddress(remoteDestinationName(),false);
      ```
      In this case, the second parameter, the Boolean “false”, indicates that the destination should not be localized to the local messaging engine, but can be anywhere on the service integration bus.
   e. Use the add method of the Vector class to add another destination name to the array.
   f. Clone the message, and modify the forward routing path in the cloned message. For example:
      ```
      clonedMessage.setForwardRoutingPath(forwardRoutingPath);
      ```
   g. Send the cloned message by using the send method in the SIMediationSession interface to send the message to the service integration bus. For example, if named "clonedMessage":

```
mediationSession.send(clonedMessage, false);
```

4. Return *true* to ensure the message passed into the handle method of the MediationHandler interface continues along the handler chain.

Example

The complete mediation function code to change the forward routing path might look like this example:

```
/* A sample mediation that clones a message
 * and sends the clone off to another destination  */

public class RoutingMediationHandler implements MediationHandler {

 public String remoteDestinationName="newdest";

 public boolean handle(MessageContext context) throws MessageContextException {
  SIMessage clonedMessage = null;
  SIMessageContext mediationContext = (SIMessageContext) context;
  SIMessage message = mediationContext.getSIMessage();
  SIMediationSession mediationSession = mediationContext.getSession();

  // Create a forward routing path that will be set on the cloned message
  Vector forwardRoutingPath = new Vector();
  SIDestinationAddressFactory destFactory =
        SIDestinationAddressFactory.getInstance();
  SIDestinationAddress dest =
        destFactory.createSIDestinationAddress(remoteDestinationName,false);
  forwardRoutingPath.add(dest);

  try {
   // Clone the message
   clonedMessage = (SIMessage) message.clone();
   // Modify the forward routing path for the clone
   clonedMessage.setForwardRoutingPath(forwardRoutingPath);
   // Send the message to the next destination in the frp
   mediationSession.send(clonedMessage, false);
  } catch (SIMediationRoutingException e1) {
   e1.printStackTrace();
  } catch (SIDestinationNotFoundException e1) {
   e1.printStackTrace();
  } catch (SINotAuthorizedException e1) {
   e1.printStackTrace();
  } catch (CloneNotSupportedException e) {
   // SIMessage should clone OK so you shouldn't need to enter this block
   e.printStackTrace();
  }
  // allow original message to continue on its path
  return true;
 }
```

SIDestinationAddress:

The SIDestinationAddress public interface represents a service integration bus destination.

The API has three methods:

- isTemporary: This method determines whether the SIDestinationAddress represents a temporary or permanent Destination, returning a Boolean value.
- getDestinationName: Method to retrieve the name of the Destination represented by this SIDestinationAddress.

- getBusName: Method to retrieve the bus name of the Destination represented by this SIDestinationAddress.

For more information about the SIDestinationAddress interface, see the SIDestinationAddress generated API information.

SIDestinationAddressFactory:

The SIDestinationAddressFactory public interface extends java.lang.Object, and creates an SIDestinationAddressFactory at static initialization that is subsequently used for the creation of all instances of SIDestinationAddress.

The interface has three methods:
- getInstance: This method gets the singleton SIDestinationAddressFactory that is used for creating SIDestinationAddress instances.
- createSIDestinationAddress: These two methods are used to create a SIDestinationAddress to represent a service integration bus destination. The first will create a SIDestination that exists only on the local service integration bus (and maybe localized to the "local" messaging engine depending on the localOnly flag). The second method is used to create a SIDestination that exists on a remote service integration bus.
-

For more information about the SIDestinationAddressFactory interface, see the SIDestinationAddressFactory generated API information.

Working with non-routing path fields in a message header:

You can work with fields in a message header that identify and affect the behavior of messages.

About this task

In addition to the routing fields (see “Setting routing addresses in a message header” on page 1009), there are a number of fields in the message header that you can work with. These fields affect important qualities and characteristics of the message, such as priority and reliability, identity, and so on. See “Message header information” for information about the equivalence of the header fields to JMS message header fields, and the methods available to work with them.

Procedure

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext **unless** you are only interested in the methods provided by MessageContext.
2. Get the SIMessage from the MessageContext object. For example, `SIMessage message = ((SIMessageContext)context).getSIMessage();`
3. Build your mediation header function in a similar way to these examples, and using the reference information in “Message header information” to help:
   a. Set the reliability of the message. For instance, `siMessage.setReliability(Reliability.ASSURED_PERSISTENT);`. In this case, the quality of service is set to the highest level.
   b. Set the time to live for a message - that is, the time, in milliseconds, that the message is allowed to remain on a queue before it is removed if it is not processed. For example, `siMessage.setRemainingTimeToLive(1000000);` will set the remaining time before the message should expire to 1000 seconds.

Message header information:

The non-routing SIMessage header fields, and the methods available to work with them, can be mapped to JMS message header fields.

Header fields

Table 107. Mapping of SIMessage header fields to JMS message header fields. The first column of the table lists the SIMessage header fields. The second column provides a brief description of the fields. The third column provides the corresponding JMS message header fields. The fourth column lists the SIMessage methods.

SIMessage header field	Field description	Corresponding JMS message header field	SIMessage methods
Priority (ReplyPriority)	Integer value 0-9, higher value is higher message priority	JMSPriority (integer)	• getPriority • setPriority • getReplyPriority • setReplyPriority
Reliability (ReplyReliability)	Specifies the reliability of message delivery. See Message reliability levels - JMS delivery mode and service integration quality of service for a description of the allowed values.	JMSDeliveryMode (string) supports two levels of reliability: PERSISTENT and NON_PERSISTENT	• getReliability • setReliability • getReplyReliebility • setReplyReliability
TimeToLive (ReplyTimeToLive, RemainingTimeToLive)	An integer that represents the time in milliseconds that a message can remain on the queue before it expires.	JMSExpiration (long) is the time of expiry, calculated as "current time" plus (+) "time-to-live".	• getTimeToLive • getReplyTimeToLive • getRemainingTimeToLive • setTimeToLive • setReplyTimeToLive • setRemainingTimeToLive
Discriminator (ReplyDiscriminator)	A string that contains a topic name that is tested by a selector rule to determine if the message should be mediated.	No corresponding JMS field	• getDiscriminator • setDiscriminator • getReplyDiscriminator • setReplyDiscriminator
RedeliveredCount	Read-only field (integer) that holds that counts each time a message is re-delivered.	JMSRedelivered (Boolean) indicates that it is likely, but not guaranteed, that the message was delivered but unacknowledged in the past.	getRedeliveredCount
ApiMessageId	A string that uniquely identifies each message sent.	JMSMessageId (string)	• getApiMessageId • setApiMessageId
CorrelationId	A string that links two messages, typically linking a request message with its response.	JMSCorrelationId (string)	• getCorrelationId • setCorrelationId
UserId	A string that represents the identity of the user sending the message.	JMSX Userid is a message property not used by WebSphere Application Server.	• getUserId • setUserId

Working with the message payload

You can work with the message payload in a pre-existing mediation handler, and transcode the message payload from one message format to another.

Before you begin

Create or open a mediation handler in an EJB project. For more information, see “Writing a mediation handler” on page 1004. You should also read the tips for successfully programming mediations in the topic Coding tips for mediations programming.

About this task

You can use this task to complete some or all of the following actions on the message payload:

- Locate the data objects within the message payload
- Convert the payload into another format
- Convert the payload into a byte array, for example if you want your mediation to log messages.

To work with the contents of a message, use the SIMessage and SIMessageContext APIs. Additionally, use SIMediationSession to provide your mediation with access to the service integration bus, to send and receive messages. For more information, see:

- Mediation programming
- “MediationHandler” on page 1015
- “SIMessageContext” on page 1015

To work with specific fields within a message, use Service Data Objects (SDO) Version 1 data graphs. For more information, see SDO data graphs. For more information about the format of supported message types, and examples of how to work with them, see “Mapping of SDO data graphs for web services messages” on page 1022.

To work with the message payload, take the following steps:

Procedure

1. Locate the point in your mediation handler where you insert the functional mediation code, in the method `handle (MessageContext context)`. The interface is MessageContext, and you should cast this to SIMessageContext unless you only want to work with the methods provided by MessageContext.
2. Retrieve the data graph of the message payload as follows:
   a. Get the SIMessage from the MessageContext object. For example:
      ```
      SIMessage message = ((SIMessageContext)context).getSIMessage();
      ```
   b. Get the message format string to determine its type. For example:
      ```
      String messageFormat = message.getFormat();
      ```
   c. Retrieve the DataGraph object from the message. For example:
      ```
      DataGraph dataGraph = message.getDataGraph();
      ```

      For more information, see SDO data graphs.
3. Optional: Locate data objects within the payload:
   a. Navigate within the graph to a named DataObject. For example, where DataObject has the name "data":
      ```
      DataObject dataObject = dataGraph.getRootObject().getDataObject("data");
      ```
   b. Retrieve information contained in the data object. For example, if the message is a text message:
      ```
      String textInfo = dataObject.getString("value");
      ```
4. Work with the fields within the message. For an example of how to do this, see “Example code for message fields” on page 1014.
5. Optional: Transcode the payload into another format:
   a. Review the topic “Transcoding between message formats” on page 1017 to understand the implications of transcoding the payload.
   b. Call the method getNewDataGraph, passing the new format as a parameter, which returns a copy of the payload in the new format. For example:
      ```
      DataGraph newDataGraph = message.getNewDataGraph(newFormat);
      ```
   c. Write the data graph in the new format back to the message using the setDataGraph method. For example:

```
message.setDataGraph(newDataGraph, newFormat);
```

6. Optional: Convert the payload into a stream of bytes:
   a. Review the topics “Transcoding a message payload into a byte array” on page 1019 and “Transcoding a byte array into a message payload” on page 1020 to understand the implications of converting between message format and byte stream, and back again.
   b. Call the method getDataGraphAsBytes, which returns a copy of the payload as a byte stream. For example:
      ```
      byte[] newByteArray = message.getDataGraphAsBytes();
      ```
   c. Call the method createDataGraph provided by the SIDataGraphFactory API, which creates a new data graph by parsing the bytes according to the format passed to the method. For example:
      ```
      DataGraph newDataGraph = SIDataGraphFactory.getInstance().createDataGraph( byteArray, format);
      ```
   d. Work with the message as a stream of bytes. For an example of how to do this, see “Example code for message fields”
7. Return *True* in your mediation code so that the MessageContext is passed to the next mediation handler in the handler list. If the return value is *False* the MessageContext will be discarded and will not be delivered to the destination.

   Note: If your mediation handler is the last handler in the handler list, and the forward routing path is empty, the message is made available to consuming applications on that destination. If the forward routing path not empty, the message is not made available to any consumers on that destination. Instead, the message is forwarded to the next destination in the routing path.

Example code for message fields

Below is an example of the code for a mediation for working with a field in a message:

```
public boolean handle(MessageContext context) throws MessageContextException {

 /* Get the SIMessage from the MessageContext object */
 SIMessage message = ((SIMessageContext)context).getSIMessage();

 /* Get the message format string */
 String messageFormat = message.getFormat();

 /* If you have a JMS TextMessage then extract the text contained in the message. */
 if(messageFormat.equals("JMS:text"))
 {
  /* Retrieve the DataGraph object from the message */
  DataGraph dataGraph = message.getDataGraph();

  /* Navigate down the DataGraph to the DataObject named 'data'. */
  DataObject dataObject = dataGraph.getRootObject().getDataObject("data");

  /* Retrieve the text information contained in the DataObject. */
  String textInfo = dataObject.get("value");

  /* Use the text information retrieved */
  System.out.println(textInfo);
 }


 /* Return true so the MessageContext is passed to any other mediation handlers
 * in the handler list */
 return true;

 }
```

The complete mediation function code for working with the message payload as a stream of bytes might look like this example:

```
  public boolean handle(MessageContext context)throws MessageContextException {

  /* Get the SIMessage from the MessageContext object */
  SIMessage message = ((SIMessageContext)context).getSIMessage();

    if (!SIApiConstants.JMS_FORMAT_MAP.equals(msg.getFormat()))
    {
      try
      {
        dumpBytes(msg.getDataGraphAsBytes());
      }
      catch(Exception e)
      {
        System.out.println("The message contents could not be retrieved due to a "+e);
      }
    }
    else
    {
      System.out.println("The bytes for a JMS:map format message cannot be shown.");
    }

    return true;
  }

  private static void dumpBytes(byte[] bytes)
  {
    // Subroutine to dump the bytes in a readable form to System.out
  }
}
```

MediationHandler:

The public interface MediationHandler has just one method: handle. This method is used by the runtime to invoke a mediation.

The method handle is called by the runtime when a message is to be mediated. The method returns Boolean *True* if the message passed into this method should continue along the handler list, otherwise *False*.

At the end of the handler list, the message is sent to the next destination on the routing path, unless the forward routing path is empty, when the message is made available to consuming applications on the current destination.

For more information about the MediationHandler interface, see the MediationHandler generated API information.

SIMessageContext:

Public interface SIMessageContext extends javax.xml.rpc.handler.MessageContext. This is the object that is required on the interface of a mediation handler. In addition to the context information that might be passed from one handler to another, it can return a reference to an SIMessage and an SIMediationSession.

The SIMessage is the service integration technologies representation of the message being processed by the MediationHandler. The SIMediationSession is a handle to the runtime resources.

The interface MessageContext abstracts the message context that is processed by a handler in the handle method. The MessageContext interface provides methods to manage a property set. MessageContext properties enable handlers in a handler chain to share processing related state.

As well as defining the method that is invoked by the mediation runtime environment, the interface might also specify properties following the Enterprise JavaBeans naming pattern, or by providing a BeanInfo class. Each property of the bean will be initialized from a single environment entry with the same name as the property. Bean properties of simple type are specified by using Java Platform, Enterprise Edition (Java EE) env-entry. If the handler has properties that are of non-simple type, then other environment definitions may be used.

The API has two methods:

- The getSIMessage method to get the service integration bus representation of the message being mediated. Read more about the SIMessage API in "SIMessage."
- The getSession method to get an SIMediationSession object that is a handle to the runtime environment. Read more about the SIMediationSession API in "SIMediationSession" on page 1017.

SIMessage:

The SIMessage interface is the public interface to a service integration bus message for use by mediations and other service integration bus components. This interface extends java.lang.Cloneable and java.lang.Serializable.

The version of Service Data Objects (SDO) supported by mediations is Version 1.

The SIMessage interface has many methods allowing you to work with message properties, header contents, routing path, metadata, and others:

- The method getDataGraph returns the SDO data graph. This contains the SIMessage payload content in a tree representation. Using the data graph, you can work directly with individual fields in the message payload. For more information about SDO data graphs, see SDO data graphs.
- You can transcode a message payload by calling the method getNewDataGraph(format). It returns a copy of the payload in the new format. You can write the new datagraph back to the message by using setDataGraph(DataGraph, format). For more information, see "Transcoding between message formats" on page 1017.
- If you want to log a message as a simple byte stream, you can retrieve the message payload as a byte array by using the method getDataGraphAsBytes. For more information about converting from data graph to bytes, and back again, see "Transcoding a message payload into a byte array" on page 1019 and "Transcoding a byte array into a message payload" on page 1020.
- There are methods to get, set, delete and clear user properties and message properties. You can also retrieve a list of user property names. For more information about working with properties, see "Working with message properties" on page 1007.
- Forward and reverse routing paths define a sequential list of intermediate bus destinations through which messages pass to reach a target bus destination. You use a routing path to apply the mediations configured on several destinations to the messages sent along the path. The following methods allow you to get and set the contents of the ForwardRoutingPath and ReverseRoutingPath for an SIMessage:
  - getForwardRoutingPath()
  - setForwardRoutingPath()
  - getReverseRoutingPath()
  - setReverseRoutingPath()

  For more information about routing paths, see Destination routing paths. For information about how to work with routing addresses, see "Setting routing addresses in a message header" on page 1009.
- If your mediation changes the content of the message, there is a risk that the message is no longer valid. If the data graph is not valid, the message cannot be sent through the service integration bus or

stored in the message store. In this case, the message is not **well formed**. A message is well formed when all the values of the message properties might be serialized, and the data graph of the message conforms to the format of the message. You can test your message by using the method isWellFormed. It returns *true* when the message contains a well formed data graph. This test has implications for performance. For more information, see Setting tuning properties for a mediation.

- You can work with the time for the message to live, measured in milliseconds from the time when the message was originally sent:
  - The methods getTimeToLive and setTimeToLive allow you to get and set the value of the TimeToLive field in the message header. A value of 0 indicates that the message will never expire.
  - The methods getRemainingTimeToLive and setRemainingTimeToLive allow you to get the remaining time in milliseconds before the message expires, and set the remaining time in milliseconds before the message should expire.

For more information about SIMessage, see the API documentation.

SIMediationSession:

The SIMediationSession public interface defines the methods for querying and interacting with the service integration bus. As well as defining the methods for working with the service integration bus, this interface also includes methods that provide information about where the mediation is invoked from, and the criteria that are applied before the message is mediated.

Both selector and discriminator control which messages are sent to the mediation, through a rule specified in a text string. The rule specified by the selector examines the header and properties of the message, whereas the discriminator examines the topic of the message. If a message contains both selector and discriminator, it must match both rules for the message to be mediated. If either the selector or the discriminator rule does not match, the message is not mediated.

The API has these methods:

- getBusName returns the name of the bus upon which the mediation is associated.
- getDestinationName returns the name of the destination with which the mediation is associated.
- getDiscriminator returns the discriminator that is defined in the mediation definition.
- getMediationName returns the name of the mediation that is being executed.
- getMessageSelector returns the message selector that is defined in the mediation definition.
- getMessagingEngineName returns the name of the messaging engine from which the mediation was invoked
- getSIDestinationConfiguration returns the SIDestinationConfiguration object associated with the destination, specified by destinationName or destinationAddress.
- receive receives an SIMessage from the service integration bus. There are four variants.
- resetIdentity changes the identity of the given message to the current run-as identity.
- send sends a copy of an SIMessage to the service integration bus, in addition to the message returned by the message interface.

See also the generated API information for SIMessageContext.

Transcoding between message formats:

A mediation can convert a message from one format to another without changing the semantic meaning of the message. This operation is referred to as transcoding a message.

The following code is an example mediation handler that transcodes a message into a new message format, providing that the message can be transcoded:

```
private static final String NEW_FORMAT = "JMS:text";

public boolean(MessageContext context) throws MessageContextException
{
  try
  {
    SIMessageContext msgCtx  = (SIMessageContext)context;
    SIMessage        msg     = msgCtx.getMessage();
    DataGraph        newDg   = msg.getNewDataGraph(NEW_FORMAT);

    msg.setDataGraph(newDg,NEW_FORMAT);
    return true;
  }
  catch(Exception e)
  {
    // Reroute the original message to the exception destination
    MessageContextException mce =
      new MessageContextException("Unable to transcode to "+NEW_FORMAT",e);
    throw mce;
  }
}
```

The table below describes which messages can be transcoded, and gives the outcome for each format pairing. Note that the abbreviation DG represents "data graph". The numbers within brackets in the table are explained as follows:

- (1) A message with format JMS: cannot have a payload. It does not carry any message data other than the message properties. If a mediation calls `getDataGraph()` on a message with format JMS:, `null` is always returned. All other message formats must have a message payload. This means that a message with format JMS: cannot be transcoded into another format. If a mediation needs to change a message with format JMS: into a message with any other format, the mediation needs to call the methods `SIDataGraphFactory.getInstance().createDataGraph(newFormat)` and `setDataGraph` on the SIMessage object to change the message contents.
- (2) `null` is always returned if a mediation calls `getDataGraph()` on a message with format JMS:
- (3) A mediation can call the method getNewDataGraph() on a message to return a copy datagraph with the same format as the message. The copy can be edited, leaving the original message unchanged. For SOAP and Beans, you can change the message model by editing the format string to change the value that follows the ":".

Table 108. Message transcoding. The table contains the different formats that can be used to convert the messages from one format to another. The table shows the transcoding options that is possible between the different formats, and it also provides the effect the conversion has on the messages.

	To JMS:	**To JMS:text**	**To JMS:bytes**	**To JMS:stream**	**To JMS:object**	**To SOAP:**	**To Bean:**
From JMS:	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)	DG=null (1)
From JMS:text	DG=null (2)	Yes (3)	Yes, bytes contain UTF-8	Yes, if text contains XML that conforms to the correct schema.	No	Yes, if message content is valid SOAP.	Yes, if message content is valid SOAP.
From JMS:bytes	DG=null (2)	Yes, but only when the bytes can correctly be interpreted as a UTF-8 string.	Yes (3)	Yes, if bytes contain XML that conforms to the correct schema.	Yes, assume that bytes are a serialized object.	Yes, if message content is valid SOAP.	Yes, if message content is valid SOAP.
From JMS:stream	DG=null (2)	Yes, text is XML transcoding.	Yes, bytes contain XML transcoding.	Yes (3)	No	No	No

Table 108. Message transcoding (continued). The table contains the different formats that can be used to convert the messages from one format to another. The table shows the transcoding options that is possible between the different formats, and it also provides the effect the conversion has on the messages.

	To JMS:	To JMS:text	To JMS:bytes	To JMS:stream	To JMS:object	To SOAP:	To Bean:
From JMS:object	DG=null (2)	No	Yes, bytes contain the object serialization.	No	Yes (3)	No	No
From SOAP:	DG=null (2)	Yes	Yes	No	No	Yes (3) - if message content matches the new WSDL.	Yes
From Bean:	DG=null (2)	Yes	Yes	No	No	Yes	Yes (3) - if message content matches the new WSDL.

XML schema definition for JMS stream messages:

This is the XML schema definition for transcoding JMS stream messages to message types.

The following XML schema definition uses the target namespace http://www.ibm.com/xmlns/prod/websphere/messaging/jms/ to express JMS stream messages in XML. Use this definition to transcode between a byte array and a message payload.

```
<xsd:schema elementFormDefault="qualified" xml:lang="EN"
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/messaging/jms"
  xmlns="http://www.ibm.com/xmlns/prod/websphere/messaging/jms"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="data" type="StreamBody"/>

  <xsd:complexType name="StreamBody">
    <xsd:sequence>
      <xsd:element name="value"
        type="streamTypes"
        minOccurs="0"
        maxOccurs="unbounded"
        nillable="true"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="character">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/>
      <xsd:maxLength value="1"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="streamTypes">
   <xsd:union memberTypes="xsd:long xsd:int xsd:short xsd:byte xsd:boolean
            xsd:float xsd:double xsd:string xsd:hexBinary character"/>
  </xsd:simpleType>
</xsd:schema>
```

Transcoding a message payload into a byte array:

You can transcode the message payload into a byte array.

For example, you might want to write a mediation handler that logs a message as a simple byte stream. You can retrieve the message payload as a byte array by using the method getDataGraphAsBytes. The table below describes the rules for transcoding an SIMessage data graph into a byte array.

Table 109. Rules for transcoding a message payload into a byte array. The first column of the table contains the datagraph formats used for transcoding a message into a byte array. The second column provides the preconditions for the datagraph format if available. The third column provides the result of the transcode. The fourth column contains the character set encodings if they are applicable for the messages.

Datagraph format	Pre-conditions	Outcome	Character set encoding
JMS:	None	Returns null.	Not applicable.
JMS:text	None	Returns the result of java.lang.String:getBytes(String charSetName) when applied to the `data/value` element of the graph, where charSetName = "UTF-8"	UTF-8
JMS:bytes	None	Returns a copy of the value of the `data/value` element of the data graph for the message.	Not applicable.
JMS:stream	None	Returns a byte buffer containing an XML serialization of the stream message according to the XML schema for stream messages.	UTF-8
JMS:object	None	Returns a copy of the value of the `data/value` element of the data graph for the message.	Not applicable.
SOAP:	If the byte array must be generated by this operation (instead of using an existing byte array available through lazy parsing) then the data graph must be valid with respect to the WSDL model.	Returns a byte buffer containing a SOAP serialization of the data graph. If the SOAP message contains an attachment, the buffer has the multipart MIME format.	Either UTF-8, or that of the source message for the graph, where logically equivalent to the graph state.
Bean:	The data graph must be valid with respect to the WSDL model. In the absence of a SOAP binding the serialization will be performed using RPC/literal encoding.	Returns a byte buffer containing a SOAP serialization of the data graph. If the Bean contains attachments then the buffer will be in multipart MIME format.	UTF-8

Transcoding a byte array into a message payload:

A mediation can transcode a byte array into a message payload without changing the meaning of the message.

A mediation can reconstruct the message payload from a byte array, for example after logging a message. To reconstruct the message, call the method `createDataGraph` provided by the SIDataGraphFactory API. This method requires a byte array and a format string and creates a new data graph by parsing the bytes according to the format passed to the method, as shown in the following example:

```
DataGraph newDataGraph = SIDataGraphFactory.getInstance().createDataGraph(byteArray, newFormat);
```

The table below describes the rules for transcoding a byte array into an SIMessage data graph.

Table 110. Rules for transcoding a byte array into an SIMessage data graph. The first column of the table contains the format arguments used for transcoding a byte array into an SIMessage data graph. The second column provides the preconditions if available for the format arguments. The third column provides the resultant SIMessage data graphs for the specified format arguments.

Format argument	Pre-conditions	Outcome
JMS:	None	Returns null
JMS:text	`java.lang.String(inputBytes, "UTF-8")` does not result in an exception.	Returns new data graph instance of format JMS:text. Value of graph at path `data/value` has value equal to `java.lang.String(inputBytes, "UTF-8")`.
JMS:bytes	`inputBytes` is not null.	Returns new data graph instance of format JMS:bytes. Value of graph at path `data/value` is a copy of the `inputBytes` byte array.
JMS:stream	Byte array is XML, and is valid with respect to the `JmsStreamBody` type of the XML schema definition.	Returns new data graph instance of format JMS:stream. Value of graph at path `data/value` has type List, containing a sequence of simple typed values according to the types and values of each of the elements in the XML document.
JMS:object	Not null **Note:** You must ensure that the byte array is a valid serialized object.	Returns new data graph instance of format JMS:object. Value of graph at path `data/value` is a copy of the `inputBytes` byte array.
SOAP:	The byte buffer contains valid SOAP with respect to the associated WSDL model.	Returns new data graph with type system defined by the WSDL referenced by the byte buffer, and values of the graph defined by the SOAP payload.
Bean:	The byte buffer contains valid Bean with respect to the associated WSDL model.	Returns new data graph with type system defined by the WSDL referenced by the byte buffer, and values of the graph defined by the Bean payload.

Web services messages overview:

To work with the data graph form of web services messages, you need to know the structure of the data graph, and how to develop code that can navigate the data graph.

The format of web services messages

WebSphere Application Server (base) supports two formats for web services messages: SOAP and enterprise beans. (These are similar to Java APIs for XML based RPC, or JAX-RPC.)

To work with web services messages, you need the following information:

- The structure of the Service Data Objects (SDO) Version 1 data graphs for web services messages. See “Mapping of SDO data graphs for web services messages” on page 1022 for more information about the data elements and the shape of the data graph.
- Reference information to help you develop code to navigate the data graphs of the messages that your program mediates. See “Mapping XML schema definitions to the SDO type system” on page 1026.
- For XML representations of the shape of each part of web services messages, sample code snippets and further information about the data graph format, see:“Web Services code example” on page 1029.

Format types

The web services message type is defined by a message format string within the message. The format string is prefixed with a domain identifier, which is either SOAP or Bean, followed by four comma-separated fields as follows:

```
SOAP:<wsdlLocation>,<serviceNameSpace>,<serviceName>,<portName>
Bean:<wsdlLocation>,<serviceNameSpace>,<serviceName>,<portName>
```

The fields are described in the following table:

Field name	Message format string	Field description
WSDL location	`<wsdlLocation>`	The URI where the WSDL for this message is located. The WSDL is deployed to the SDO repository, by using this location as the key.
Service namespace	`<serviceNamespace>`	Service namespace and Service name uniquely identify the Service definition within the WSDL.
Service name	`<serviceName>`	Service name and Service namespace uniquely identify the correct Service definition within the WSDL.
Port name	`<portName>`	Locates the Port definition within the Service, giving the PortType and Binding information that is required for message processing.

Mapping of SDO data graphs for web services messages:

The structure of web services messages is described by the Service Data Objects (SDO) Version 1 data graphs for web services messages.

Overall layout of a web services message

A web services message is described by a format string and three metadata fields: operationName, messageName, and messageType. The payload of the message is split across three other sections: headers, attachments and the body.

The Info node is the top-level of the SDO data graph for all web services messages. The following table describes the Info node properties and associated types.

Property name	Property type	Property description
operationName	`java.lang.String`	Identifies the WSDL operation that is associated with the message. If the data access service cannot identify the message, this field is null. See “Identifying web services messages” on page 1023.
messageName	`java.lang.String`	Identifies the WSDL message that is associated with the message. If the data access service cannot identify the message, this field is null. See “Identifying web services messages” on page 1023.

Property name	Property type	Property description
messageType	`java.lang.String`	Identifies WebService type of message instance. This field can have the values: *input*, *output*, *fault*, *ambiguous*. If the data access service cannot identify the message, this field is null. See “Identifying web services messages.”
headers	`java.util.List` of data objects.	Contains a list of header entry data objects. Each SOAP header in the message results in a header entry in this list. See “Message header layout ” on page 1024.
attachments	`java.util.List` of data objects.	Contains a list of attachment entry data objects. For SOAP messages with attachments, each MIME part in the message (except the MIME part containing the SOAP envelope) is mapped to an entry in this list. See “Message attachment layout” on page 1024.
body	`commonj.sdo.DataObject`	A nested data object, which represents the body of the SOAP envelope. See “Message body layout” on page 1025.

Identifying web services messages

Processing of messages depends on whether the messages have WSDL definitions. The minimum amount of information required for processing without WSDL is “SOAP:”. The minimum amount of information required for processing with WSDL is: “SOAP:location,namespace,service,port”. If the format string does not include all five of these fields, the SOAP data access service attempts to process the message without WSDL.

- Processing messages without WSDL definitions: If the format string does not include full WSDL information, the SOAP data access service processes the message without attempting to match the message against definitions in WSDL. As a result, operationName and messageName are set to null, and messageType is set only when processing a fault message.
- Processing messages with WSDL definitions: If the format string includes <WSDL location>,<Service namespace>,<Service name>, and <Port name>, the SOAP and Beans data access services process the message by using the WSDL definitions of the service.

  Note: In either of the following circumstances, SOAP message processing fails once it has supplied all the required WSDL information:
  - the SOAP data access service fails to locate the WSDL
  - the WSDL fails to corroborate the message

When the SOAP data access service processes a SOAP request or reply message, it tries to match it against the message definitions in the WSDL. Usually there is one matching definition, and the operationName, messageName, and messageType are filled in appropriately. If there is more than one matching definition, the data access service selects a message definition, fills in the operationName and messageName. and sets the messageType to *ambiguous*.

When processing fault messages, identification is slightly different. In all cases the messageType is set to *fault*. If the message matches a unique fault definition in the WSDL, the operationName and messageName properties are also set.

Message header layout

The list of headers has two types of entry, header entry or bound header entry, depending on whether the header is based on part of the message or not.

The header entry type is used to handle headers that meet either of the following criteria:

- the header is part of the message that is modeled in WSDL
- the header is part of the message that is not modeled in WSDL, but is based on a part of the message

For a model of this header, see “Header entry.”

The bound header entry type is used when the SOAP binding for the message has bound a part of the body into a MIME attachment. (This occurs when you use a <MIME:content> element to bind a part of the message to an attachment.) For consistent mediation programming, all of the body data is stored in the body node in the graph. Unlike the normal attachment entry, a bound attachment entry is placed into the attachments list. The bound attachment entry contains the MIME meta-data for the attachment and the name of the message part that contains the data taken from this attachment. This allows mediations designed to process attachments to locate the data in the body part of the data graph. For a model of this attachment see “Bound header entry.”

Header entry

Property name	Property type	Property description
mustUnderstand	`java.lang.Boolean`	Carries the value from the mustUnderstand attribute on the SOAP header, if present.
actor	`java.lang.String`	Carries the value from the actor attribute on the SOAP header, if present.
any	`commonj.sdo.Sequence`	Container for the contents of the SOAP header.

Bound header entry

Property name	Property type	Property description
mustUnderstand	`java.lang.Boolean`	Carries the value from the mustUnderstand attribute on the SOAP header, if present.
actor	`java.lang.String`	Carries the value from the actor attribute on the SOAP header, if present.
messagePart	`java.lang.String`	Contains the name of the message part that carries the data from this message header.

Message attachment layout

Message attachments are handled in a similar way to headers, and instances of them populate the attachments list in the Info node.

There are two types of attachment entry to handle MIME attachments: attachment entry and bound attachment entry.

Attachment entry is for general attachments: see “Attachment entry” on page 1025.

Bound attachment entry includes <MIME:content> elements that bind a part of the body into a MIME attachment. If you are programming a mediation, you must know how to locate the data within the graph. For consistent mediation programming, the attachment data is placed in the message body, referred to by the part name in the header entry, which includes the other MIME metadata. For a model of this attachment, see "Bound attachment entry."

Attachment entry

Property name	Property type	Property description
contentType	`java.lang.String`	Carries the contentType from the MIME part that is represented by the attachment entry.
contentTransferEncoding	`java.lang.String`	Carries the contentTransferEncoding from the MIME part that is represented by the attachment entry.
contentId	`java.lang.String`	Carries the contentId from the MIME part that is represented by the attachment entry.
data	`byte[]`	Carries the content of the MIME element as a byte array.

Bound attachment entry

Property name	Property type	Property description
contentType	`java.lang.String`	Carries the contentType from the MIME part that is represented by the attachment entry.
contentTransferEncoding	`java.lang.String`	Carries the contentTransferEncoding from the MIME part that is represented by the attachment entry.
contentId	`java.lang.String`	Carries the contentId from the MIME part that is represented by the attachment entry.
messagePart	`java.lang.String`	Contains the name of the message part that carries the data from this attachment.

Message body layout

The layout of the data object in the body is defined by the service WSDL. The type of the data object is derived from the message definition in the WSDL. The data object has one property for each part in the message definition. The layout of each message part follows the convention for mapping XML Schema into SDO. See "Web Services code example" on page 1029 for more information.

Web services fault message

If the message is a fault message, the messageType field in the Info node of the graph is set to "fault", and the message body has the following properties:

Property name	Property type	Property description
faultcode	`javax.xml.namespace.QName`	Carries the faultcode value from the SOAP Fault element
faultstring	`java.lang.String`	Carries the faultstring value from the SOAP Fault element
faultactor	`java.lang.String`	Carries the faultactor value from the SOAP Fault element

Property name	Property type	Property description
detail	`commonj.sdo.DataObject`	Carries the content within the detail child of the SOAP Fault element

Note: The detail element definition uses element and attribute wildcards, so the content of the detail data object contains a Sequence. See “Web Services code example” on page 1029 for more information.

Mapping XML schema definitions to the SDO type system:

Each XML schema type is mapped to an SDO type. Use this mapping to help you develop code to navigate the data graphs of the messages that your program mediates.

The version of Service Data Objects (SDO) supported by mediations is Version 1.

XML schemas can be embedded in the WSDL sections that describe the message parts and SOAP headers. However the SOAP header description is more likely to be available as a separate schema, in which case you should load it into the SDO repository where it can be used at run time to process any message with a matching header.

Schema to Java class mapping

Each XML schema complex type is mapped to an SDO type. This means that an element with a complex type is represented by an instance of an SDO data object. The type has a property for each element, attribute, or wildcard that is contained in the schema type definition.

The instance contains a value for each property that has been set. If the property is mapped from a schema complex type, the value is another SDO data object. If the property is mapped from a schema simple type, the value is an instance of a Java class, as shown in the following table.

Table 111. Schema types and corresponding Java classes. The first column of the table lists the schema types. The second column lists the corresponding Java classes. The third column refers you to one of the two notes that is given at the end of the table for your reference.

Schema type	Java class	Notes
anyURI	java.lang.String	
base64Binary	byte[]	See Note 2
boolean	java.lang.Boolean/ boolean	See Note 1
byte	java.lang.Byte / byte	See Note 1
date	java.lang.String	
dateTime	java.lang.String	
decimal	java.math.BigDecimal	
double	java.lang.Double / double	See Note 1
duration	java.lang.String	
ENTITIES	java.util.List	
ENTITY	java.lang.String	
float	ava.lang.Float / float	See Note 1
gDay	java.lang.String	
gMonth	java.lang.String	
gMonthDay	java.lang.String	
gYear	java.lang.String	
gYearMonth	java.lang.String	
hexBinary	byte[]	See Note 2

Table 111. Schema types and corresponding Java classes (continued). The first column of the table lists the schema types. The second column lists the corresponding Java classes. The third column refers you to one of the two notes that is given at the end of the table for your reference.

Schema type	Java class	Notes
ID	java.lang.String	
IDREF	java.lang.String	
IDREFS	java.util.List	
int	java.lang.Integer / int	See Note 1
integer	java.math.BigInteger	
language	java.lang.String	
long	java.lang.Long / long	See Note 1
Name	java.lang.String	
NCName	java.lang.String	
negativeInteger	java.math.BigInteger	
NKTOKENS	java.util.List	
NMTOKEN	java.lang.String	
nonNegativeInteger	java.math.BigInteger	
nonPositiveInteger	java.math.BigInteger	
normalisedString	java.lang.String	
NOTATION	javax.xml.namespace.QName	
positiveInteger	java.math.BigInteger	
QName	javax.xml.namespace.QName	
short	java.lang.Short / short	See Note 1
string	java.lang.String	
time	java.lang.String	
token	java.lang.String	
unsignedByte	java.lang.Short / short	See Note 1
unsignedInt	java.lang.Long / long	See Note 1
unsignedLong	java.math.BigInteger	
unsignedShort	java.lang.Integer / int	See Note 1

Notes:

1. SDO automatically converts primitives (int, long and so on) into objects, as needed. This means that you can use a mixture of the specialized methods (getInt, setInt, getLong, setLong) as well as the generic get and set methods.
2. As byte arrays are mutable, you can update the value without setting it back onto the data object. However, when this occurs, the data object may not be aware of implicit update. When working with byte array values, always use the setBytes() method to explicitly update the data object.

Working with global elements and attributes

When a schema is mapped to SDO, a special SDO type - typically called 'DocumentRoot' - is defined. This type is a container for all the global elements and attributes in the schema. Whenever you have to locate an SDO property for a global element or attribute, you should locate the ‘DocumentRoot' type and then locate the appropriate property within it.

The following schema defines the layout of web services messages. By comparing this schema with the information in “Mapping of SDO data graphs for web services messages” on page 1022 you can see the schema to SDO mapping in action.

```
<?xml version="1.0"?>
<xsd:schema
    targetNamespace="http://www.ibm.com/ns/2004/05/webservices/messagemodel"
    xmlns:tns="http://www.ibm.com/ns/2004/05/webservices/messagemodel"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

 <xsd:import namespace="http://schemas.xmlsoap.org/soap/envelope/"/>

    <xsd:element name="Info">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="operationName" nillable="true" type="xsd:string"/>
          <xsd:element name="messageName"   nillable="true" type="xsd:string"/>
          <xsd:element name="messageType"   nillable="true" type="xsd:string"/>
          <xsd:element name="headers"       type="tns:HeaderEntryType"     maxOccurs="unbounded"/>
          <xsd:element name="attachments"   type="tns:AttachmentEntryType" maxOccurs="unbounded"/>
          <xsd:element name="body"          type="tns:BodyType"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="BodyType" abstract="true"/>

    <xsd:complexType name="HeaderEntryType" abstract="true"/>

    <xsd:complexType name="AttachmentEntryType" abstract="true"/>

    <xsd:complexType name="SOAPFaultBody">
      <xsd:complexContent>
        <xsd:extension base="tns:BodyType">
          <xsd:sequence>
            <xsd:element name="faultcode" type="xsd:QName"/>
            <xsd:element name="faultstring" type="xsd:string"/>
            <xsd:element name="faultactor" type="xsd:anyURI" minOccurs="0"/>
            <xsd:element name="detail" type="soap:detail" minOccurs="0"/>
    </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="SOAP_1_1_HeaderEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:HeaderEntryType">
          <xsd:sequence>
            <xsd:element name="mustUnderstand" nillable="true" type="xsd:boolean"/>
            <xsd:element name="actor"          nillable="true" type="xsd:anyURI"/>
            <xsd:any/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="SOAP_1_1_BoundHeaderEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:HeaderEntryType">
          <xsd:sequence>
            <xsd:element name="mustUnderstand" nillable="true" type="xsd:boolean"/>
            <xsd:element name="actor"          nillable="true" type="xsd:anyURI"/>
            <xsd:element name="messagePart"    type="xsd:string"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
```

```
    </xsd:complexType>

    <xsd:complexType name="MIMEAttachmentEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:AttachmentEntryType">
          <xsd:sequence>
            <xsd:element name="contentType"             type="xsd:string"/>
            <xsd:element name="contentTransferEncoding" type="xsd:string"/>
            <xsd:element name="contentId"               type="xsd:string"/>
            <xsd:element name="data"                    type="xsd:base64Binary"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="BoundMIMEAttachmentEntryType">
      <xsd:complexContent>
        <xsd:extension base="tns:AttachmentEntryType">
          <xsd:sequence>
            <xsd:element name="contentType"             type="xsd:string"/>
            <xsd:element name="contentTransferEncoding" type="xsd:string"/>
            <xsd:element name="contentId"               type="xsd:string"/>
            <xsd:element name="messagePart"             type="xsd:string"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="UnknownBodyType">
      <xsd:complexContent>
        <xsd:extension base="tns:BodyType">
          <xsd:sequence>
            <xsd:any/>
          </xsd:sequence>
          <xsd:attribute name="encodingStyle" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

Web Services code example:

This Web Services Description Language (WSDL) example and code snippets show how to access fields within a Web services message for programming a mediation.

Web services message definition

This topic contains an example of a web services message. The example is characterized in WSDL, an XML-based language that is used to describe the services a business offers and how those services might be accessed.

This topic shows how to program mediations to work with different parts of a web services message, which are described with the Service Data Objects (SDO) Version 1 representation in “Mapping of SDO data graphs for web services messages” on page 1022. For each part of the message, there is an XML description of the message, representing its SDO data graph. Each XML description is accompanied by code snippets that illustrate how to work with that part of the message.

Note: In the following example, the SOAP header schema is included in the WSDL. Alternatively, it can be included as a separate schema in the SDO repository.

This is a WSDL description of the message that is used as an example for subsequent code snippets:

companyInfo web service message description

```
<wsdl:definitions targetNamespace="http://example.companyInfo"
 xmlns:tns="http://example.companyInfo"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:wsdlmime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
  <xsd:schema elementFormDefault="qualified"
   targetNamespace="http://example.header">

   <xsd:element name="sampleHeader">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name="priority" type="xsd:int"/>
     </xsd:all>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>


  <xsd:schema elementFormDefault="qualified"
   targetNamespace="http://example.companyInfo">

   <xsd:element name="getCompanyInfo">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name="tickerSymbol" type="xsd:string"/>
     </xsd:all>
    </xsd:complexType>
   </xsd:element>

   <xsd:element name="getCompanyInfoResult">
    <xsd:complexType>
     <xsd:all>
      <xsd:element name="result" type="xsd:float"/>
     </xsd:all>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>


 </wsdl:types>

 <wsdl:message name="getCompanyInfoRequest">
   <wsdl:part name="part1" element="tns:getCompanyInfo"/>
 </wsdl:message>

 <wsdl:message name="getCompanyInfoResponse">
  <wsdl:part name="part1" element="tns:getCompanyInfoResult"/>
  <wsdl:part name="part2" type="xsd:string"/>
  <wsdl:part name="part3" type="xsd:base64Binary"/>
 </wsdl:message>

 <wsdl:portType name="CompanyInfo">
  <wsdl:operation name="GetCompanyInfo">
   <wsdl:input message="tns:getCompanyInfoRequest"
               name="getCompanyInfoRequest"/>
   <wsdl:output message="tns:getCompanyInfoResponse"
```

```
                name="getCompanyInfoResponse"/>
  </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="CompanyInfoBinding" type="tns:CompanyInfo">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="GetCompanyInfo">
   <wsdlsoap:operation soapAction=""/>
   <wsdl:input name="getCompanyInfoRequest">
    <wsdlsoap:body use="literal"/>
   </wsdl:input>
   <wsdl:output name="getCompanyInfoResponse">
    <wsdlsoap:body use="literal"/>
   </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="CompanyInfoService">
  <wsdl:port binding="tns:CompanyInfoBinding" name="SOAPPort">
   <wsdlsoap:address location="http://somewhere/services/CompanyInfoService"/>
  </wsdl:port>
 </wsdl:service>

</wsdl:definitions>
```

Working with the info node

This is an example of a simple SOAP request:

```
<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
```

You can access the properties of the info node (see “Overall layout of a web services message” on page 1022) using code snippets such as this:

```
 // Get the info node (a child of the graph root object)
 DataObject rootNode = graph.getRootObject();
 DataObject infoNode = rootNode.getDataObject("Info");

  // Query the operationName, and messageType.
  String opName = infoNode.getString("operationName");
  String type   = infoNode.getString("messageType");
```

Working with a header

This is an example of a SOAP request including a header:

```
<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Header>
        <example:sampleHeader
          env:mustUnderstand='1'
          xmlns:example='http://example.header'>
            <example:priority>4</example:priority>
        </example:sampleHeader>
    </env:Header>
```

```
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
```

For a description of the properties of the header entry with a list of headers, see “Header entry” on page 1024. To work with a header entry and its properties, use code such as this:

```
   // Get the info node (a child of the graph root object)
    DataObject rootNode = graph.getRootObject();
    DataObject infoNode = rootNode.getDataObject("Info");

    // Access the list of headers
    List headerEntries = infoNode.getList("headers");

    // Get the first entry from the list
    DataObject headerEntry = (DataObject) headerEntries.get(0);

    // Query the mustUnderstand property of the header entry
    boolean mustUnderstand = headerEntry.getBoolean("mustUnderstand");

    // Get the Sequence that holds the content of the header entry
    Sequence headerContent = headerEntry.getSequence("any");

    // Get the first piece of content from the Sequence
    DataObject header = (DataObject) headerContent.getValue(0);

    // Read the priority from the header
    int priority = header.getInt("priority");

    // Shorthand for the above, using SDO path expressions that start
    // from the info node.
    mustUnderstand = infoNode.getBoolean("headers[1]/mustUnderstand");
    priority       = infoNode.getInt("headers[1]/any[1]/priority");
```

Working with an attachment

This is an example of a SOAP request including an XML attachment:

```
Content-Type: multipart/related; start="<start>"; boundary="boundary"


--boundary
Content-Type: text/xml
Content-Transfer-Encoding: 7bit
Content-ID: <start>

<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
--boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-ID: <myAttachment>

<info>Some attached information</info>
--boundary--
```

For a description of the properties of the attachment entry with byte array, see “Attachment entry” on page 1025. To work with a header entry and its properties, use code such as this:

```
// Get the info node (a child of the graph root object)
  DataObject rootNode = graph.getRootObject();
  DataObject infoNode = rootNode.getDataObject("Info");

  // Access the list of attachments
  List attachmentEntries = infoNode.getList("attachments");

  // Get the first entry from the list
  DataObject attachmentEntry = (DataObject) attachmentEntries.get(0);

  // Query the contentId property of the header entry
  String contentId = attachmentEntry.getString("contentId");

  // Get the data contained in the attachment
  byte[] data = attachmentEntry.getBytes("data");
```

Working with the message body

This is an example of a simple SOAP request:

```
<env:Envelope
  xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:ns1='http://example.companyInfo'>
    <env:Body>
        <ns1:getCompanyInfo>
            <ns1:tickerSymbol>IBM</ns1:tickerSymbol>
        </ns1:getCompanyInfo>
    </env:Body>
</env:Envelope>
```

To see the properties of the body, see “Message body layout” on page 1025. To work with the contents of the body, use code such as this:

```
  // Get the info node (a child of the graph root object)
  DataObject rootNode = graph.getRootObject();
  DataObject infoNode = rootNode.getDataObject("Info");

  // Get hold of the body node
  DataObject bodyNode = infoNode.getDataObject("body");

  // Get hold of the data object for the first part of the body
  DataObject part1Node = bodyNode.getDataObject("part1");

  // Query the tickerSymbol
  String ticker = part1Node.getString("tickerSymbol");

  // Shorthand for the above, using a SDO path expression that
  // starts from the info node.
  ticker = infoNode.getString("body/part1/tickerSymbol");
```

JMS formats:

To write code that can access the different JMS message types, you need to know how each message type is mapped to SDO, and how to retrieve the message format string from the message.

Format types

Each JMS message type is defined by a message format string within the message. You can retrieve the format string by using the code snippet in the following example . The table shows the mapping of message format strings to Service Data Objects (SDO):

JMS Message type	Message format string	Mapping to SDO[1]
JMS Bytes message	`JMS:bytes`	See “JMS Formats - bytes”
JMS Text message	`JMS:text`	See “JMS Formats - text”
JMS Stream message	`JMS:stream`	See “JMS formats - Stream” on page 1035
JMS Object message	`JMS:object`	See “JMS Formats - object” on page 1035
JMS Map message	`JMS:map`	The retrieval or construction of SDO data graphs for JMS map messages is not supported.

[1] The version of SDO supported by mediations is Version 1.

This code snippet is an example of how to retrieve the message format string from the message:

```
String format = siMsg.getFormat();
if (format.equals ....
```

JMS Formats - bytes:

Your program can retrieve the payload of a JMS bytes message by mapping the body of the message to an SDO data graph representing the message.

Bytes body

You can retrieve the payload of a JMS bytes message as a Java byte array (`byte[]`). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named "data", and that data object in turn contains a property named "value". In JMS bytes messages, the value property might be accessed as a Java byte array.

You can access the data within the data graph with code such as this:

```
SIMessage siMsg;
String format = siMsg.getFormat();
if (format.equals("JMS:bytes")) {
  DataGraph graph = siMsg.getDataGraph();
  byte[] payload = graph.getRootObject().getBytes("data/value");
}
```

JMS Formats - text:

Your program can retrieve the payload of a JMS text message by mapping the body of the message to an SDO data graph representing the message.

Text body

You can retrieve the payload of a JMS text message as a Java string value (java.lang.String). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named "data", and that data object in turn contains a property named "value". In JMS text messages the value property may be accessed as a Java string value.

You can access the data within the data graph with code such as this:

```
SIMessage siMsg;
String format = siMsg.getFormat();
if (format.equals("JMS:text")) {
  DataGraph graph = siMsg.getDataGraph();
  String payload = graph.getRootObject().getString("data/value");
}
```

JMS formats - Stream:

Your program can retrieve the payload of a JMS stream message by mapping the body of the message to an SDO data graph representing the message.

Stream body

You can retrieve the payload of a JMS Stream message as a Java list value (java.util.List). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named "data", and that data object in turn contains a property named "value". For a JMS Stream message the value property might be accessed as a List value. The member functions of the List interface can be used to access the individual objects within the JMS Stream message instance. (Note that the JMS standard places constraints on the kinds of objects that might be placed in a Stream message.)

You can access the data within the data graph with code such as this:

```
}SIMessage siMsg;
String format = siMessage.getFormat();
if (format.equals("JMS:stream")) {
  DataGraph graph = siMsg.getDataGraph();
  List payload = graph.getRootObject().getList("data/value");
  int streamLength = payload.size();
  if (streamLength > 0) {
    Object item1 = payload.get(0);
    // You can also access items directly, for example: (for the_same_ value)
    item1 = graph.getRootObject().get("data/value[1]");
  }
}
```

JMS Formats - object:

Your program can retrieve the payload of a JMS object message by mapping the body of the message to an SDO data graph representing the message.

Object body

You can retrieve the payload of a JMS object message as a Java byte array (`byte[]`). First, you must retrieve a data graph representing the message from the SIMessage instance. As is common to all data graphs representing JMS messages, the root data object of the graph contains a property named "data", and that data object in turn contains a property named "value". For a JMS object message the value property might be accessed as a Java byte array. The original Object instance that the payload represents might be reconstructed from the byte array.

You can access the data within the data graph with code such as this:

```
SIMessage siMsg;
String format = siMsg.getFormat();
if (format.equals("JMS:object")) {
  DataGraph graph = siMsg.getDataGraph();
  byte[] payload = graph.getRootObject().getBytes("data/value");
  if(payload != null) {
    // Need to deserialize to recover original object
```

```
    ObjectInputStream in =
      new ObjectInputStream(new ByteArrayInputStream(payload));
    Object obj = in.readObject();
  }
}
```

Writing a routing mediation

Use this topic to create a mediation that chooses a particular forward route for a message.

Before you begin

For an introduction to using mediations with the service integration bus, see Learning about mediations. For details of how to install a mediation into WebSphere Application Server and associate it with a bus destination, see Working with mediations.

This topic assumes that you are familiar with using a Java Platform, Enterprise Edition (Java EE) session bean development environment such as the assembly tools or IBM Rational Application Developer.

About this task

A routing mediation is a mediation application that contains a routing handler. You associate a routing mediation with a service integration bus destination, and use the mediation to choose a particular route from a range of available routes. For example when you create a new outbound service configuration or modify an existing outbound service configuration you can apply a port selection mediation to choose a particular outbound port from the range of ports that are available to the outbound service.

To create a routing mediation, use a Java Platform, Enterprise Edition (Java EE) session bean development environment to complete the following steps:

Procedure

1. Create an empty mediation handler project. This creates the project, and creates the handler class that implements the handler interface. For detailed instructions on how to do this, see Writing the mediation handler.
2. Use the mediation pane on the EJB descriptor to define the handler class as a mediation handler.

   Note: When you do this, you specify a name by which the mediation handler list is known. Make a note of this name, for later reference when you create the mediation in the bus.
3. Add the routing function to the handler. Before you begin, review Adding mediation function to handler code, in particular its subtopic Working with message context. Add import statements to your handler class, and modify the handle method by adding your routing code. Specify the routing destination by adding that destination to the front of the forward routing path list. The forward routing path list is available from the message context. For example:

```
import javax.xml.rpc.handler.MessageContext;
import com.ibm.websphere.sib.mediation.handler.MediationHandler;
import com.ibm.websphere.sib.mediation.handler.MessageContextException;
import com.ibm.websphere.sib.mediation.messagecontext.SIMessageContext;
import com.ibm.websphere.sib.SIMessage;
import com.ibm.websphere.sib.SIDestinationAddress;
import com.ibm.websphere.sib.SIDestinationAddressFactory;
import java.util.List;
public class RouteMediationHandler implements MediationHandler {

 public boolean handle(MessageContext ctx) throws MessageContextException {
  SIMessageContext siCtx = (SIMessageContext) ctx;
  SIMessage msg = siCtx.getSIMessage();
  List frp = msg.getForwardRoutingPath();
  try {
```

```
   SIDestinationAddress destination =
    SIDestinationAddressFactory
     .getInstance()
     .createSIDestinationAddress(
     "RoutingDestination", //this is the name of the target destination
     false);
   frp.add(0, destination);
  } catch (Exception e) {
   return false;
  }
  msg.setForwardRoutingPath(frp);
  return true;
 }

}
```

For more information about the service integration technologies classes, including the mediation handler and message context classes, see the Generated API documentation - Application programming interfaces .

4. Export the routing mediation enterprise application.

What to do next

You are now ready to install your mediation into WebSphere Application Server and associate it with a bus destination, as described in Working with mediations.

Writing a mediation that maps between attachment encoding styles

Use this topic to create a mediation that maps from SOAP Messages with Attachments encoding style to WS-I Attachments Profile Version 1.0 encoding style.

Before you begin

For an introduction to using mediations with the service integration bus, see Learning about mediations. For details of how to install a mediation into WebSphere Application Server and associate it with a bus destination, see Working with mediations.

This topic assumes that you are familiar with using a Java Platform, Enterprise Edition (Java EE) session bean development environment such as the assembly tools or IBM Rational Application Developer.

The example mediation given in this topic is based upon the WSDL examples that are given in Supporting bound attachments: WSDL examples

About this task

You can use a mediation to map from a SOAP Messages with Attachments encoding of a message to WS-I Attachments Profile Version 1.0 encoding. The WSDL definition is the same in both cases, so if you create a mediation that rewrites the Content ID values to match the Version 1.0 conventions then the message is encoded by service integration technologies according to Version 1.0 rules.

To create a mapping mediation, use a Java Platform, Enterprise Edition (Java EE) session bean development environment to complete the following steps:

Procedure

1. Create an empty mediation handler project. This creates the project, and creates the handler class that implements the handler interface. For detailed instructions on how to do this, see Writing the mediation handler.
2. Use the mediation pane on the EJB descriptor to define the handler class as a mediation handler.

Note: When you do this, you specify a name by which the mediation handler list is known. Make a note of this name, for later reference when you create the mediation in the bus.

3. Add the mapping function to the handler. Before you begin, review Adding mediation function to handler code. Here is an example of mediation handler code that rewrites the Content ID values to match the Version 1.0 conventions:

```
int uuidBase = 0;
DataObject root = SIMessage.getDataGraph().getRootObject();
List attachments = root.getList("info/attachments");
Iterator entries = attachments.iterator();
while(entries.hasNext()) {
  DataObject entry = (DataObject) entries.next();
  if(entry.getType().equals("BoundMIMEAttachmentEntryType")) {
    String newContentId = entry.getString("messagePart") + "=" +
                          Integer.toString(uuidBase++) +
                          "@some.domain";
  }
}
```

Note: For messages that use a SOAP with attachments reference (swaref) or some other URI mechanism to refer to the attachments, the URI values might also have to be updated to match the new Content ID values. However such mechanisms are usually used to refer to unbound attachments.

For more information about the service integration technologies classes, including the mediation handler classes, see the Generated API documentation - Application programming interfaces .

4. Export the mapping mediation enterprise application.

What to do next

You are now ready to install your mediation into WebSphere Application Server and associate it with a bus destination, as described in Working with mediations.

Choosing a target service and port through a routing mediation

Write a routing mediation and configure it to select a target service and port. If you have migrated a web services gateway that included a routing filter, use this task to recreate the equivalent functions by using a mediation.

About this task

One gateway service can map to one or more target services, and (for outbound target services) each target service can have one or more ports as defined in the outbound service WSDL. Without a routing mediation there is no point in mapping multiple targets because the gateway always picks the default destination. If you want to map multiple targets, you must write a routing mediation then configure it, for each gateway service, to select the target service and target port.

You associate routing mediations with service integration bus destinations. To pick a target service, you associate the mediation with the gateway service destination. To pick an outbound service port, you associate the mediation with the outbound service destination.

Use the steps described in Writing a routing mediation to help you write a mediation application that contains a routing handler, install it into WebSphere Application Server and associate it with a destination.

Using durable subscriptions

You use *durable subscriptions* for publish/subscribe messaging. A durable subscription can be used to preserve messages published on a topic while the subscriber is not active.

About this task

If there is no active subscriber for a durable subscription, JMS retains the subscription messages until they are received by the subscriber, or until they expire, or until the durable subscription is deleted. This enables subscriber applications to operate disconnected from the JMS provider for periods of time, and then reconnect to the provider and process messages that were published during their absence.

Each JMS durable subscription is identified by a subscription name (*subName*), which is defined when the durable subscription is created. A JMS connection also has an associated client identifier (*clientID*), which is used to associate a connection and its objects with the list of messages (on the durable subscription) that is maintained by the JMS provider for the client. The subName assigned to a durable subscription must be unique within a given client ID.

If an application needs to receive messages published on a topic while the subscriber is inactive, it uses a *durable subscriber*.

In normal operation there can be at most one active (connected) subscriber for a durable subscription at a time. However, when running inside an application server it is possible to clone the application server for failover and load-balancing purposes. In this case, a cloned durable subscription can have multiple simultaneous consumers.

For information about durable subscriptions, see the JMS 1.1 Specification (for example, section 9.3.3 “Using Durable Subscriptions”).

The following operations for durable subscriptions are in addition to the usual JMS operations, such as to first look up a connection factory and a JMS destination, and to create a connection and session.

The following are the main operations for using durable subscriptions:

- Creating a new durable subscription
- Reconnecting to an existing durable subscription
- Unsubscribing (deleting) a durable subscription

Procedure

- Define the Durable Subscription Home This property must be set on the JMS connection factory if durable subscriptions are to be created using connections created from this connection factory. The value is the name of the messaging engine where all durable subscriptions accessed through this connection are managed.

  You can also set the Durable Subscription Home on the JMS topic destination, which enables a single connection to access durable subscriptions on more than one messaging engine.

  To be able to create durable subscriptions, the property on the connection factory must not be null (the default). Setting a value of null or empty string on the property of a destination indicates that the value specified on the connection factory should be inherited.
- Creating a new durable subscription A durable TopicSubscriber can be created by a Session or by a TopicSession.

  Having performed the normal setup, an application can create a durable subscriber to a destination. To do this, the client program creates a durable TopicSubscriber, by using session.createDurableSubscriber. The name *subName* is used as an identifier of the durable subscription.

```
session.createDurableSubscriber( Topic topic,
        java.lang.String subName,
        java.lang.String messageSelector,
        boolean noLocal);
```

Alternatively, you can use the two-argument form of this operation, which takes only a topic and name (*subName*) as parameters. This alternative form invokes the four-argument operation with null as the messageSelector and false as the noLocal parameters.

```
session.createDurableSubscriber( Topic topic, java.lang.String subName);
```

A JMS durable subscription is created with a unique identifier of the form *clientID*+"##"+*subName*. The characters ## should not be used in the clientID or subName if the JMS connection is to use a durable subscription.

Handling exceptions. The following JMS exceptions can be thrown for the reasons listed in the exception messages:

- InvalidDestination - The name of this durable subscription (*clientID*+"##"+*subName*) clashes with an existing destination.
- IllegalState - The method was invoked on a closed connection.
- IllegalState - This destination is not accepting consumers. This probably means that there is already an active subscriber for this durable subscription.
- InvalidDestination - The mediation named in the parameters cannot be found.
- InvalidDestination - The destination cannot be found.
- JMSSecurity - The user does not have authorization to perform this operation.
- JMSException - Errors occurred in the MsgStore, Comms or Core layers.

- Reconnecting to an existing durable subscription To reconnect to a topic that has an existing durable subscription, the subscriber application calls `session.CreateDurableSubscriber` again, using the same parameters that it used to originally create the durable subscription. However, consider the following important restrictions:
  - The subscriber must be attached to the same connection.
  - The destination and subscription name must be the same as in the original method call.
  - If a message selector was specified, it must be the same as in the original method call.

  By calling createDurableSubscriber again, the subscriber application reconnects to the topic, and receives any messages that arrived while the subscriber was disconnected.
- Unsubscribing (deleting) a durable subscription To unsubscribe (delete) a durable subscription to a topic, the subscriber application calls `session.unsubscribe(java.lang.String name)`.

  Do not call the unsubscribe method to delete a durable subscription if there is a TopicConsumer currently consuming messages from the topic.

Sending web service messages directly over the bus from a JAX-RPC client

Use this task to send web service messages over a bus by retargeting the JAX-RPC client.

About this task

Java API for XML-based remote procedure calls (JAX-RPC) client applications send and receive web service request and response messages. JAX-RPC client applications that use the IBM JAX-RPC run-time environment can do this in a number of different ways, depending on the bindings in the WSDL document that they are developed against, and the configuration data that is used at run time.

For an introduction to basic JAX-RPC programming concepts, including the JAX-RPC client and server programming models, see Getting Started with JAX-RPC.

If you want to use a JAX-RPC client to send messages over the service integration bus, you have two choices:

- Use a SOAP binding (SOAP over HTTP or SOAP over JMS), and pass messages indirectly through an endpoint listener to an inbound service. You would do this if you had SOAP-specific JAX-RPC handlers that must run in the client application context.
- Pass messages directly into the service integration bus at a destination by “retargeting” the JAX-RPC client application as described in this topic.

  Note: There are currently limitations regarding the Java types used by services that are retargeted through a JAX-RPC client application.

Retargeting involves setting the following two values into the client application deployment descriptor, or specifying them dynamically at run time from within the client application:

- The *binding namespace* is set to indicate that the client uses the messaging bus directly.
- The *endpoint address* is set to include the particular destination and (optionally) the format of messages that the client uses.

The destination also needs to be configured so that it knows the port type of messages that the JAX-RPC client is using. There are two ways to achieve this:

- Create an outbound service. An outbound service represents an externally-provided web service. In this case, requests from the JAX-RPC client pass through the service destination and are then sent on to the service provider defined by the outbound service configuration.
- Create an inbound service. An inbound service represents a service provided somewhere within or beyond the messaging bus. You can create an inbound service on any existing destination. The creation of an inbound service associates a WSDL port type with the destination. When retargeting to a destination with an inbound service, the client application needs to specify both the destination name and inbound service name, because it is possible to configure more than one inbound service against a single destination. In this case, requests from the JAX-RPC client pass through the destination and then onwards through the service integration bus depending on routing that is done at the initial destination.

To have web service messages sent directly to a destination using a JAX-RPC client, complete the following steps:

Procedure

1. Create the JAX-RPC client application.
2. Create the outbound service or inbound service with which you want the JAX-RPC client application to exchange messages.
3. Use the administrative console to access the port information for your JAX-RPC client application, as described in “Configuring web services client bindings” on page 2208 and Web services client port information.
4. Override the default SOAP binding for your JAX-RPC client application. Change the binding namespace to `http://www.ibm.com/ns/2004/02/wsdl/mp/sib`
5. Override the endpoint that your JAX-RPC client application uses to send web service requests. The new endpoint should use the sib: URL syntax and include either the outbound service destination name, or both the inbound service name and its corresponding destination name.

What to do next

After you change the *binding namespace*, any JAX-RPC handler lists that were configured for the retargeted port are ignored. For clients that are developed against WSDL with a SOAP binding, retargeting directly to the bus causes the handlers to be ignored. However if the client is developed against the non-bound WSDL for the service, retargeting to the bus is not considered to be changing the binding namespace, and so the handler information is retained. In this case the JAX-RPC handlers are called with the SDOMessageContext subclass.

Associated reference information:

- “sib: URL syntax”

sib: URL syntax

The `sib:` URL has the following syntax:

`sib:/[destination|path]?property_1=value_1&property_2=value_2&...`

where:

- Square brackets (“[ ]”) indicate that a parameter is optional.
- Transport type is `sib:`, followed by either `/destination` to specify destination type or `/path` to specify a forward routing path, followed by a “query string” that contains one or more properties. The permitted properties are described in the subsequent sections of this topic.

Required properties

The following properties are required. They are used to specify the destination for the request.

Note: All destination names must be fully-qualified. That is, they must include the name of the service integration bus as well as the destination name itself. Use the syntax *bus:destination*. If a bus or destination name contains a colon or comma, wrap the name in double quotation marks (“”). If it contains a double quotation mark, repeat the quotation mark.

destinationName
The destination name.

path The forward routing path, in the form of a sequence of destination names separated by commas.

replyDestinationName
The name of the destination to be used for the reply.

inboundService
The name of the inbound service that identifies the specific attachment that the requester application uses. You can omit this value if the destination is a service destination with an associated outbound service configuration, because in that case the requester is attaching to the outbound service through the service destination.

timeout
The time the requester waits for a response. The default value is 60 seconds. A zero value indicates an unlimited wait.

Service integration technologies-related properties

The following properties are optional. If you do not specify a value for a property, then the default value is used. For more information regarding the permitted values for these properties, see the generated API information for the SIMessage interface.

reliability
The reliability of the request message.

timeToLive
The amount of time (in milliseconds) before the request times out. A zero value indicates that the request never times out.

Note: The **timeout** property (see the required properties) is the time delay after which the requester application blocks the application thread that is waiting for a response to a request and response operation. The **time to live** and **replyTimeToLive** optional properties indicate how long the request and reply messages should be processed by the messaging

engines. This does not include the processing time at the service implementation. You would therefore usually set the timeout to be the sum of the request and response times to live, plus some amount for the service processing time.

priority
The priority of the request message.

user
The user ID required to access the request destination.

password
The password required to access the request destination.

replyReliability
The reliability of the reply message.

replyTimeToLive
The amount of time (in milliseconds) before the reply times out. A zero value indicates that the reply never times out.

replyPriority
The priority of the reply message.

Other properties

You can also include user-defined properties in the URL. These properties must be named with a `user.` prefix. For example:

```
sib:/destination?destinationName=myBus:myDestination & reliability=assured & user.customData=XYZ
```

After the request is sent, the URL itself is available within the message properties, named `inbound.url`.

Chapter 24. Developing Session Initiation Protocol (SIP) applications

This page provides a starting point for finding information about SIP applications, which are Java programs that use at least one Session Initiation Protocol (SIP) servlet written to the JSR 116 specification.

SIP is used to establish, modify, and terminate multimedia IP sessions including IP telephony, presence, and instant messaging.

Developing SIP applications

A SIP application is a set of SIP servlets packaged in a SIP application archive file (SAR).

About this task

A SIP servlet is an application component managed by the SIP container that performs SIP signaling. The programming and deployment models are analogous to web servlets and therefore will be mapped to the WebSphere administrative model accordingly. It is possible to include web servlets in a SAR file (along with the required web.xml deployment descriptor) to create what is known as a converged application. See JSR 116 for details on SIP applications, servlets, converged applications, and status codes.

Developing SIP applications that support PRACK

A SIP response to an INVITE request can be final or provisional. Final responses are always sent reliably, but provisional responses typically are not. For cases where you need to send a provisional response reliably, you can use the PRACK (Provisional response acknowledgement) method.

Before you begin

For you to be able to develop applications that support PRACK, the following criteria must be met:

- The client that sends the INVITE request must put a 100rel tag in the Supported or the Require header to indicate that the client supports PRACK.
- The SIP servlet must respond by invoking the sendReliably() method instead of the send() method to send the response.

About this task

PRACK is described in the following standards:

- RFC 3262 (“Reliability of Provisional Responses in the Session Initiation Protocol (SIP)”), which extends RFC 3261 (“SIP: Session Initiation Protocol”), adding PRACK and the option tag 100rel.
- Section 6.7.1 (“Reliable Provisional Responses”) of JSR 116 (“SIP Servlet API Version 1.0”).

Procedure

- For an application acting as a proxy, do this:
  - Make your application generate and send a reliable provisional response for any INVITE request that has no tag in the To field.
- For an application acting as a user agent client (UAC), do this:
  - Make your application add the 100rel tag to outgoing INVITE requests. The option tag must appear in either the Supported header or the Require header.
  - Within your application's doProvisionalResponse(...) method, prepare the application to create and send PRACK requests for incoming reliable provisional responses. The application must create the PRACK request on the response's dialog through a SipSession.createRequest(...) method, and it must set the RAck header according to RFC 3262 Section 7.2 (“RAck”).

  - The application that acts as an UAC will not receive doPrack( ) methods. The UAC sends INVITE and receives Reliable responses. When the UAC receives the Reliable response, it sends PRACK a request to the UAS and receives a 200 OK on the PRACK so it should next implement doResponse( ) in order to receive it.
- For an application acting as a user agent server (UAS), do this:
  - If an incoming INVITE request requires the 100rel tag, trying to send a 101-199 response unreliably by using the send() method causes an Exception.
  - Make the application declare a SipErrorListener to receive noPrackReceived() events when a reliable provisional response is not acknowledged within 64*T1 seconds, where T1 is a SIP timer. Within the noPrackReceived() event processing, the application should generate and send a 5*xx* error response for the associated INVITE request per JSR 116 Section 6.7.1.
  - Make the application have at most one outstanding, unacknowledged reliable provisional response. Trying to send another one before the first's acknowledgement results in an Exception.
  - Make sure that the application enforces the RFC 3262 offer/answer semantics surrounding PRACK requests containing session descriptions. Specifically, a servlet must not send a 2*xx* final response if any unacknowledged provisional responses contained a session description.

Setting up SIP application composition

The JSR 116 standard for SIP applications states in section 2.4 that multiple applications may be invoked for the same SIP request. The process of setting up applications to comply with this standard is called application composition.

About this task

Application composition requires that implementations use a cascaded services model. The cascaded services model requires that service applications triggered on the same host are triggered in sequence, as if the triggering occurred on different hosts. Therefore responses flow upstream and hit applications in the reverse order of the corresponding requests.

The JSR 116 standard does not specify how to implement application composition, thus there are many ways to comply with this standard. For WebSphere Application Server, composition of the application depends on the deployed application order, and on the order of mapping rules within the deployment descriptor of each application.

- For an initial incoming request, the SIP container tries each potential rule in order. When the container finds the n^{th} match, the container invokes the corresponding servlet.
- If the servlet must proxy the request, the container scans the rules again to search for additional matches. When the container finds the $(n+1)^{th}$ match, the container invokes the corresponding servlet.
- Any servlet in the same application as the previously invoked servlet is excluded from the matching process. No servlet can be invoked twice for the same SIP request.

You can specify load on start-up priority. The `<load-on-startup>` in the `sip.xml` defines the order in which servlets are initialized on startup. If this value is lower than zero, the servlets are initialized when the first request is matched to them according to matching rule and composition order. Zero is a legitimate weight for startup initialization order. If this tag does not exist or if it contains a negative value, the servlet does not initialize at startup.

You should also add `<load-on-startup>` to the same tag in the `web.xml` if you are changing it manually. It is the WebContainer that loads servlets (and siplets), and it looks only at the `web.xml`. When deploying a SAR, only the `sip.xml` needs to be changed. The `web.xml` is automatically constructed correctly after deployment.

The load-on-startup tag embedded in the SIP deployment descriptor tag for a servlet dictates the order that the application is loaded on start up of the server. It does not dictate the order that an application gets called when the application is a member of an application composition chain that matches rules to process a new message coming in.

The starting weight for applications and their modules is specified in the `deployment.xml` file. The order in which modules pickup requests on composition is evaluated by applications weight first and then modules weight. The following steps can be completed in any order to specify applications weight or modules weight from the administrative console.

Procedure

1. To specify the applications (EARs) weight, expand **Enterprise Applications** > ***applicationName*** > **Startup Behavior** and set the startup order.
2. To specify the modules (WARs) weight, expand **Enterprise Applications** > ***applicationName*** > **Manage Modules** and set the starting weight.
3. Restart the changed applications.

Example

Specifying load-on-startup priority example:

sip.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app
PUBLIC "-//Java Community Process//DTD SIP Application 1.0//EN"
"http://www.jcp.org/dtd/sip-app_1_0.dtd">
<sip-app>
 <display-name>SIPSampleProxy</display-name>

 <servlet>
  <servlet-name>SIPSampleProxy</servlet-name>
  <servlet-class>sipes.test.container.proxy.SIPSampleProxy</servlet-class>
  <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
  <servlet-name>SIPSampleProxy</servlet-name>
  <pattern>
   <equal>
    <var>request.uri.user</var>
    <value>SIPSampleProxy</value>
   </equal>
  </pattern>
 </servlet-mapping>

 <proxy-config>
  <sequential-search-timeout>1000</sequential-search-timeout>
 </proxy-config>
 <session-config>
  <session-timeout>12</session-timeout>
 </session-config>
</sip-app>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp">
 <display-name>SIPSampleProxy</display-name>
 <servlet>
  <servlet-name>SIPSampleProxy</servlet-name>
  <display-name>SIPSampleProxy</display-name>
  <servlet-class>sipes.test.container.proxy.SIPSampleProxy</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>SIPSampleProxy</servlet-name>
  <url-pattern>/SIPSampleProxy</url-pattern>
 </servlet-mapping>
 <welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
```

```
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
 </welcome-file-list>
</web-app>
```

Specifying starting weight example:

The following example is for a stand-alone server.

deployment.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <appdeployment:Deployment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:appdeployment="http://www.ibm.com/websphere/appserver/schemas/5.0/appdeployment.xmi"
xmi:id="Deployment_1137951186883">
- <deployedObject xmi:type="appdeployment:ApplicationDeployment" xmi:id="ApplicationDeployment_1137951186883"
deploymentId="0" startingWeight="1" binariesURL="$(APP_INSTALL_ROOT)/OrangeNode08Cell/SipContainerTestSuite.ear"
useMetadataFromBinaries="false" enableDistribution="true" createMBeansForResources="true" reloadEnabled="false"
appContextIDForSecurity="href:OrangeNode08Cell/SipContainerTestSuite"
filePermission=".*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755" allowDispatchRemoteInclude="false"
allowServiceRemoteInclude="false">
  <targetMappings xmi:id="DeploymentTargetMapping_1137951186883" enable="true" target="ServerTarget_1137951186883" />
  <classloader xmi:id="Classloader_1137951186883" mode="PARENT_FIRST" />
- <modules xmi:type="appdeployment:WebModuleDeployment" xmi:id="WebModuleDeployment_1137951186883"
deploymentId="1" startingWeight="10000" uri="sipunit.war">
  <targetMappings xmi:id="DeploymentTargetMapping_1137951186884" target="ServerTarget_1137951186883" />
  <classloader xmi:id="Classloader_1137951186884" /> </modules>
  <properties xmi:id="Property_1137951186883" name="validateinstall" value="warn" /> </deployedObject>
  <deploymentTargets xmi:type="appdeployment:ServerTarget" xmi:id="ServerTarget_1137951186883"
name="server1" nodeName="OrangeNode10" /> </appdeployment:Deployment>
```

SIP servlets

This topic describes SIP servlets.

The SIP Servlet 1.0 specification (JSR 116) is standardized through Java Specification Request (JSR) 116. The idea behind the specification is to provide a Java application programming interface (API) similar to HTTP servlets, which provides an easy-to-use SIP programming model. Like the popular HTTP servlet programming model, some flexibility is limited to optimize ease-of-use and time-to-value.

However, the SIP Servlet API is different in many ways from HTTP servlets because the protocol is so different. While SIP is a request-response protocol, there is not necessarily only one response to every one request. This complexity and a need for a high performing solution meant that it was easier to make the SIP servlets natively asynchronous. Also, unlike HTTP servlets, the programming model for SIP servlets sought to make client requests easy to create alongside the other logic being written because many applications act as a client or proxy to other servers or proxies.

SipServlet requests

Like HTTP servlets, each SIP servlet extends a base javax.servlet.sip.SipServlet class. All messages come in through the service method, which you can extend. However, because there is not a one-to-one mapping of requests to responses in SIP, the suggested practice is to extend the doRequest or doResponse methods instead. When extending the doRequest or doResponse methods, it is important to call the extended method for the processing to complete.

Each request method, which the specification must support, has a doxxx method just like HTTP. In HTTP, methods such as doGet and doPost exist for GET and POST requests. In SIP, doInvite, doAck, doOptions, doBye, doCancel, doRegister, doSubscribe, doNotify, doMessage, doInfo, and doPrack methods exist for each SIP request method.

Unlike an HTTP servlet, SIP servlets have methods for each of the response types that are supported. So, SIP servlets include the doProvisionalResponse, doSuccessResponse, doRedirectResponse, and doErrorResponse responses. Specifically, the provisional responses (1xx responses) are used to indicate status, the success responses (2xx responses) are used to indicate a successful completion of the transaction, the redirect responses (3xx responses) are used to redirect the client to a moved resource or entity, and the error responses (4xx, 5xx, and 6xx responses) are used to indicate a failure or a specific

error condition. These types of response messages are similar to HTTP, but because the SIP Servlet programming model includes a client programming model, it is necessary to have responses handled programmatically as well.

Clarifications of JSR 116

JSR 289 has made some clarifications to JSR 116, as follows:

- JSR 289 Section 4.1.3: Contact Header Field
- JSR 289 Section 5.2: Implicit Transaction State
- JSR 289 Section 5.8: Accessibility of SIP Servlet Messages

SIP SipServletRequest and SipServletResponse classes

The SipServletRequest and SipServletResponse classes are similar to the HttpServletRequest and HttpServletResponse classes.

SipServletRequest and SipServletResponse classes

Each class gives you the capability to access the headers in the SIP message and manipulate them. Because of the asynchronous nature of the requests and responses, this class is also the place to create new responses for the requests. When you extend the doInvite method, only the SipServletRequest class is passed to the method. To send a response to the client, you must call the createResponse method on the Request object to create a response. For example:

```
protected void doInvite(SipServletRequest req) throws
 javax.servlet.ServletException, java.io.IOException {

 //send back a provisional Trying response
 SipServletResponse resp = req.createResponse(100);
resp.send();
```

Because of their asynchronous nature, SIP servlets can seem complicated. However, something as simple as the previous code sample sends a response to a client.

Here is a more complex example of a SIP servlet. With the following method included in a SIP servlet, the servlet blocks all of the calls that do not come from the example.com domain.

```
protected void doInvite(SipServletRequest req) throws
 javax.servlet.ServletException, java.io.IOException {

//check to make sure that the URI is a SIP URI
      if (req.getFrom().getURI().isSipURI()){
                  SipURI uri = (SipURI)req.getFrom.getURI();
                  if (!uri.getHost().equals("example.com")) {
                     //send forbidden response for calls outside domain
                     req.createResponse(SipServletResponse.SC_FORBIDDEN).send();
                     return;
                  }
               }
               //proxy all other requests on to their original destination
               req.getProxy().proxyTo(req.getRequestURI);
        }
```

SIP SipSession and SipApplicationSession classes

Possibly the most complex portions of the SIP Servlet 1.0 specification are the SipSession and SipApplicationSession classes.

SIP SipSession and SipApplicationSession classes

Both of these classes have some useful purposes and can act as the primary place to store data in applications that are designed for distributed or highly available environments.

The SipSession class is the best representative of a specific point-to-point communication between two entities and is the closest to the HttpSession object. Because historically no proxying or forking existed for the HTTP request in HTTP servlets, the need for something higher than a single point-to-point session did not exist. However, even HTTP users can see the growing need for this type of function since portlets began essentially forking HTTP requests. The SIP users expect the proxying and forking activities that require multiple layers of SIP session management. The SipSession class is the lowest point-to-point layer.

The SipApplicationSession class represents the higher layer of SIP session management. One SipApplicationSession class can own one or more SipSession objects. However, each SipSession class can be related to one SipSession object only. The SipApplicationSession class also supports the attachment of any number of other protocol sessions. Currently, only HTTP sessions are supported by any implementations. The SipApplicationSession class has a getSessions method, which takes the requested protocol type as an argument.

You might find it useful for many applications to combine HTTP and SIP. For example, you might use this approach to tie together HTTP and SIP sessions to monitor a phone call or to start a phone call through a rich HTTP graphical user interface.

Example: SIP servlet simple proxy

This is a servlet example of a simple proxy.

Simple proxy

```
import java.io.IOException;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.sip.Proxy;
import javax.servlet.sip.SipFactory;
import javax.servlet.sip.SipServlet;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServletResponse;
import javax.servlet.sip.SipSession;
import javax.servlet.sip.SipURI;
import javax.servlet.sip.URI;


public class SimpleProxy extends SipServlet implements Servlet {


 final static  private String  SHUTDOWN_KEY = new String("shutdown");
 final static  private String  STATE_KEY = new String("state");
 final static  private int   INVITE_RECEIVED = 1;

 /* (non-Java-doc)
  * @see javax.servlet.sip.SipServlet#SipServlet()
  */
 public SimpleProxy() {
  super();
 }

 /* (non-Javadoc)
  * @see javax.servlet.sip.SipServlet#doInvite(javax.servlet.sip.SipServletRequest)
  */
 protected void doInvite(SipServletRequest request) throws ServletException,
   IOException {

  //log("SimpleProxy: doInvite: TOP");

  try {
   if (request.isInitial() == true)
   {
    // This should cause the sip session to be created. This sample only uses the session on receiving
    // a BYE but the Tivoli performance viewer can be used to track the creation of calls by viewing the
```

```
   // active session count.
   Integer state = new Integer(INVITE_RECEIVED);
   SipSession session = request.getSession();
   session.setAttribute(STATE_KEY, state);
            log("SimpleProxy: doInvite: setting attribute");

   Proxy proxy = request.getProxy();

   SipFactory sipFactory = (SipFactory) getServletContext().getAttribute(SIP_FACTORY);
         if (sipFactory == null) {
             throw new ServletException("No SipFactory in context");
         }

         String callingNumber = request.getTo().toString();
         if (callingNumber != null)
         {
             String destStr = format_lookup(callingNumber);
             URI dest = sipFactory.createURI(destStr);

             //log("SimpleProxy: doInvite: Proxying to dest URI ="  + dest.toString());

          if (((SipURI)request.getRequestURI()).getTransportParam() != null)
           ((SipURI)dest).setTransportParam(((SipURI)request.getRequestURI()).getTransportParam());

          proxy.setRecordRoute(true);
    proxy.proxyTo(dest);
         }
   else
   {
             //log("SimpleProxy: doInvite: Request is invalid. Did not contain a To: field.");
    SipServletResponse sipresponse = request.createResponse(400);
    sipresponse.send();
   }
  }
  else
  {
         //log("SimpleProxy: doInvite: target refresh, let container handle invite");
   super.doInvite(request);
  }
 }
 catch (Exception e){
  e.printStackTrace();
 }
}

/* (non-Javadoc)
 * @see javax.servlet.sip.SipServlet#doResponse(javax.servlet.sip.SipServletResponse)
 */
protected void doResponse(SipServletResponse response) throws ServletException,
  IOException {
  super.doResponse(response);

  // Example of using the session object to store session state.
 SipSession session = response.getSession();
 if (session.getAttribute(SHUTDOWN_KEY) != null)
 {
  //log("SimpleProxy: doResponse: invalidating session");
  session.invalidate();
 }
}

/* (non-Javadoc)
 * @see javax.servlet.sip.SipServlet#doBye(javax.servlet.sip.SipServletRequest)
 */
protected void doBye(SipServletRequest request) throws ServletException,
  IOException {
```

```
  SipSession session = request.getSession();
  session.setAttribute(SHUTDOWN_KEY, new Boolean(true));

        //log("SimpleProxy: doBye: invalidate session when responses is received.");
  super.doBye(request);
 }

 protected String format_lookup(String toFormat){
  int start_index = toFormat.indexOf('<') + 1;
  int end_index = toFormat.indexOf('>');

  if(start_index == 0){
   //don't worry about it
  }
  if(end_index == -1){
   end_index = toFormat.length();
  }

  return toFormat.substring(start_index, end_index);
 }
}
```

Example: SIP servlet SendOnServlet class

The SendOnServlet class is a simple SIP servlet that would perform the basic function of being called on each INVITE and sending the request on from there.

SendOnServlet class

Function could easily be inserted to log this invite request or reject the INVITE based on some specific criteria.

```
package com.example;
import java.io.IOException;
import javax.servlet.sip.*;
import java.servlet.ServletException;
public class SendOnServlet extends SipServlet {
  public void doInvite(SipServletRequest req)
       throws ServletException, java.io.IOException {
     //send on the request
     req.getProxy().proxyTo(req.getRequestURI);
  }
}
```

The doInvite method could be altered to do something such as reject the invite for some specific criteria simply. In the following example, all requests from domains outside of example.com will be rejected with a Forbidden response.

```
   public void doInvite(SipServletRequest req)
throws ServletException, java.io.IOException {
if (req.getFrom().getURI().isSipURI()){
    SipURI uri = (SipURI)req.getFrom.getURI();
    if (!uri.getHost().equals("example.com")) {
  //send forbidden response for calls outside domain
req.createResponse(SipServletResponse.SC_FORBIDDEN, "Calls outside example.com not accepted").send();
return;
    }
 }
 //proxy all other requests on to their original destination
 req.getProxy().proxyTo(req.getRequestURI());
   }


SendOnServlet deployment descriptor:
<sip-app>
    <display-name>Send-on Servlet</display-name>
    <servlet>
```

```
        <servlet-name>SendOnServlet</servlet-name>
        <servlet-class>com.example.SendOnServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>SendOnServlet</servlet-name>
        <pattern>
              <equal>
                <var>request.method</var>
                <value>INVITE</value>
              </equal>
        </pattern>
    </servlet-mapping>
</sip-app>
```

Example: SIP servlet Proxy servlet class

Proxy servlet class

After the initial INVITE, this application will be called on every subsequent SIP message. For each Request and Response, this class will simply print out the action and who it is to or from.

```
package com.example;
import java.io.IOException;
import javax.servlet.sip.*;
import java.servlet.ServletException;
public class ProxyServlet extends SipServlet {
  public void doInvite(SipServletRequest req)
       throws ServletException, java.io.IOException {
     //get the Proxy
     Proxy p=req.getProxy();
     //turn on supervised mode so that all events come through us
     //The default on this is true but it is set to emphasize the function.
     p.setSupervised(true);
     //set record route so we see the ACK, BYE, and OK
     p.setRecordRoute(true);
     //proxy on the request
     p.proxyTo(req.getRequestURI());
  }
public void doRequest(SipServletRequest req)
     throws ServletException, java.io.IOException {
   System.out.println(req.getMethod()+" Request from "+req.getFrom().getDisplayName());
   super.doRequest(req);
}
public void doResponse(SipServletResponse resp)
     throws ServletException, java.io.IOException {
   System.out.println(resp.getReasonPhrase()+" Response from "+resp.getTo().getDisplayName());
   super.doResponse(resp);
}
}

Proxy deployment descriptor
<sip-app>
    <display-name>ProxyServlet</display-name>
    <servlet>
        <servlet-name>ProxyServlet</servlet-name>
        <servlet-class>com.example.ProxyServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ProxyServlet</servlet-name>
        <pattern>
              <equal>
                <var>request.method</var>
                <value>INVITE</value>
```

```
            </equal>
        </pattern>
    </servlet-mapping>
</sip-app>
```

JSR 289 overview

Version 8.5 includes support for SIP Servlet Specification 1.1, also referred to as Java Specification Request (JSR) 289.

The SIP Servlet Specification provides the Java API standards for Session Initiation Protocol (SIP). JSR 289 is an update to the existing SIP Servlet specification that addresses new requirements determined by industry users.

SIP is a signaling protocol used for creating, modifying, and terminating IP communication sessions such as telephony and presence applications. SIP is not limited to voice communication and can mediate any type of communication session, such as multimedia.

The following is a brief description of new features available in the JSR 289 specification.

- Application router for application selection

  Application routing enables developers to build complex services out of smaller applications. On initial requests the container calls the application router to determine which application to invoke based on the type of request. The application router is the central hub for selecting application order. See the topic on configuring a SIP application router for more information.

- Annotation-based programming

  Annotations provide a fast way to develop applications by embedding metadata directly in applications. For example, you can use the `@SipServlet` annotation to indicate that a class is a SIP servlet. The `@SipApplication` is a package level annotation. All servlets in the package belong to the same application unless the servlet uses `@SipServlet(applicationName)`. For more information on annotations, see section 18 of the JSR 289.

- Converged applications

  JSR 289 provides a new, standardized mechanism for building converged applications. A converged application contains SIP servlet components and other Java EE components, like HTTP servlets and enterprise beans. The specification includes two new classes to support convergence.

  – `ConvergedHttpSession` is an extension to HttpSession for converged applications.

  – `SipSessionUtil` handles session management for converged applications.

  For more information on converged applications, see section 13 of the JSR 289.

- Back-to-back user agent (B2BUA) APIs

  JSR 289 simplifies the B2BUA pattern in applications with the use of the B2BUA helper class. The B2BUA is a frequently used application pattern. The B2BUA acts as an endpoint for two or more dialogs and forwards requests and responses between those dialogs. The B2BUA helper has the ability to create a copy of an incoming request. It also automatically maintains links between sessions on both sides of the B2BUA. For more information on B2BUAs, see section 12 of the JSR 289.

Note: Support for the Session Initiation Protocol (SIP) session key-based targeting mechanism that is described in JSR 289 section 15.11.2 is only supported in stand-alone environments. The session key-based targeting mechanism is not supported in clustered environments. Alternatively, you can use one of the other targeting mechanisms that are described in JSR 289 within a clustered environment:

- Encode the URI mechanism that is described in section 15.11.3.
- Join and replace the targeting mechanism that is described in section 15.11.4.

SIP application router:

The SIP application router is used by the SIP container to select the order in which applications are run within the container.

The SIP container can invoke multiple applications in order to deploy a complete service or function. This modular and compositional approach makes it easier for application developers to develop new applications. The modular applications can be more easily combined and managed, while individual application implementations remain independent.

The application router is responsible for selecting the correct applications in the correct order to service an incoming message. An application router is required for a container to function, but it is a separate logical entity from the container. The application router is based on the JSR 289 specification. See the specification for more details about the application router function.

The default application router (DAR) can be configured with a standard configuration file, which is supplied to the container through a SIP container custom property, as defined in JSR 289. The DAR configuration file can also be uploaded in the administrative console for each target of the DAR.

Application routing, also referred to as application composition, can be handled in a number of ways:

- Specify the order in which the applications should run using the administrative console.
- Upload a custom application router implementation class either by specifying the path of the Java archive (JAR) file containing the application router implementation and provider through the console or adding it to the class path. A specific provider can be defined with a SIP container custom property.
- Configure the DAR by uploading its properties file and providing its location through a system property.
- Use an interactive wizard to generate a DAR configuration file.

Restriction: WebSphere Application Server has a default way of sorting the order of SIP applications invocation using the Startup behavior settings. The sorting order is based on the application weight. This weighting policy only applies if you do not specify a DAR property file and no custom application router has been associated with the server or cluster.

Note: If CEA features are used, the CEA system application requires special consideration when enabled on the same server or cluster as a custom application router. To deploy an application router and still maintain the capabilities of the CEA system application, use one of the following two options:

- Only enable CEA on an isolated server or cluster that includes no custom application router.
- Make sure the custom application router routes all CEA specific messages to the CEA system application. To do this, the developer of the application router must check the mappings that are defined in the sip.xml file associated with the CEA system application. The sip.xml file associated with the CEA system application can be found in the directory path at *app_server_root/systemApps*.

  The following information explains how to configure a custom application router to route to the commsvc system application. The examples show a custom application router configuration with and without the commsvc application.

  First, here is an example configuration without commsvc:

```
INVITE: ("TestB2bua", "DAR:To", "NEUTRAL", "", "NO_ROUTE", "0")
```

  The first element after the INVITE is the display name of the test application, and this one-line application router routes b2bua calls to the application successfully. With the preceding application router configured on the SIP container, however, CEA Web collaboration attempts fail.

  To enable routing to the CEA system application, just clone the routing element and change the application name in the second element instance:

```
INVITE: ("TestB2bua", "DAR:To", "NEUTRAL", "", "NO_ROUTE", "0"),("commsvc", "DAR:To", "NEUTRAL", "", "NO_ROUTE", "0")
```

  This action ensures that CEA messages are routed correctly.

Tuning considerations using the JSR 289 Application Router with multiple applications:

This topic describes performance adjustments and considerations using the JSR 289 Application Router™ with multiple applications.

Note: This topic references one or more of the application server log files. As a recommended alternative, you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files on distributed and IBM i systems. You can also use HPEL in conjunction with your native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.

When you deploy more than one application, you might see the following errors in the log files when heavy SIP protocol traffic exists for a single application server or cluster of servers:

- Unexpected and excessive SIP application 503 Server Unavailable error messages
- Proxy and Server overload errors

Note: These error messages do not occur when you deploy one application.
The proxy server and Session Initiation Protocol (SIP) containers are not synchronized when they are tracking the amount of messages that are flowing through the system. Using the application router, multiple messages might be routed between applications. These messages cause container message counters to increment even though the messages do not flow through the proxy server.

You can diagnose this problem when you have the following conditions:

- Heavy SIP protocol traffic exists.
- Multiple applications are deployed on a single node or cluster.

Check the proxy server, the application server SystemOut.log log files, or both for an unexpected overload condition that is detected at the proxy server, the application server, or both. Also, look for 503 Server unavailable messages that are logged from the SIP application.

Resolving the problem

Messages are shared between applications at the SIP container before they are sent to the proxy server. To avoid these error messages and a decrease in SIP performance, tune the SIP containers to consider the additional SIP messages that are generated when using the application router with multiple applications. Complete the following steps in the administrative console to tune the SIP containers:

1. Expand **Servers** > **Server Types** and click **WebSphere application servers** > ***server_name***
2. Under **Container Settings**, expand **SIP Container Settings** and click **SIP container**.
3. Increase the **Maximum messages per averaging period** value to compensate for the anticipated increase in messages that are generated by the SIP application router.
4. Increase the **Maximum application sessions** value to compensate for the increased **Maximum messages per averaging period** value.

The proxy server cannot detect the amount of messages that are generated at the server. However, modifications to the following settings might increase the messaging capacity at the containers for the number of applications that are deployed per container.

Table 112. DAR and CAR SIP container tuning values.

This table lists the DAR and CAR SIP container tuning values for the number of applications that are deployed per container.

SIP Container	Single Deployed SIP Application	Three Deployed SIP Applications
Maximum messages per averaging period	value = 26640	value = 79920
Maximum application sessions	value = 36000	value = 96000

Note: The values for the **Maximum messages per averaging period** and **Maximum application sessions** fields depend on the processing power, memory, and the deployed application. Use the values for these fields as listed in the SIP container settings topic and adjust them to meet the needs of your environment.

Developing applications that use the Asynchronous Invocation API

You can use the Asynchronous Invocation API to transfer events that require processing in the context of a Session Initiation Protocol (SIP) application session to any server in a cluster based on the related application session ID. The Asynchronous Invocation API transfers the event task to the correct server.

Before you begin

Read the API documentation for information on the following asynchronous work classes:

- com.ibm.websphere.sip.AsynchronousWork
- com.ibm.websphere.sip.AsynchronousWorkListener

For more information about the API classes, see the Reference section in the information center and click APIs - Application Programming Interfaces to view a list of the product API specifications.

About this task

When running code outside of a SIP thread, application developers can use the Asynchronous Invocation API to create an object, and configure the server to run that object either on a different thread in the same container, or on a different server, if that is where the session exists.

The following example shows the class structure for the AsynchronousWork class, which is the abstract base class that is extended when using the API.

```
public abstract class AsynchronousWork implements Serializable
{
private String sessionId;
public AsynchronousWork(String sessionId)
{
this.sessionId = sessionId;
....
}
public void dispatch (AsynchronousWorkListener listener)
{
....
}
public abstract Serializable doAsyncTask();
}
```

Procedure

1. Extend the abstract class AsynchronousWork with the SIP related code. The extended implementation of the doAsyncTask() method is invoked on the target server that contains the SipApplicationSession, and whose ID was set in the constructor that implements the AsynchronousWork class. The implementation class must pass the session ID to the base class by calling super in the constructor.

```
public class MyClass extends AsynchronousWork
{
String _sessionId;

public MyClass(String sessionId) {
super(sessionId);
_sessionId = sessionId;
}

// This code is invoked on the target machine or thread
public Serializable doAsyncTask() {
// Application code goes here; for instance:
appSession = sessionUtils.getApplicationSession(_sessionId);
appSession.createRequest().....

Serializable myResponse = new MyResponse();
myResponse.setStatus(200);
return (myResponse);
}
}
```

2. To receive information about the task completion, implement the AsynchronousWorkListener class, as in the following example. The code in these methods are invoked on the source server.

```
public class MyListener implements AsynchronousWorkListener
{
public void workCompleted(Serializeable myResponse)
{
....
}
public void workFailed(int reasonCode, String reason)
{
}
}
```

3. To invoke the asynchronous call; for example, when you receive the proprietary message, use this sample code as an example.

```
public void onMyMessage()
{
// Obtain the session ID from the message or
// somewhere else
String sessionId = obtainIdFromMessage();

// Create the runnable
MyClass myClass = new MyClass(sessionId);

// Create the listener
MyListener myListener = new MyListener();

// Dispatch it
myClass.dispatch(myListener);
}
```

Results

The SIP container ensures the task is invoked in the correct server and on the correct thread, so that the application can avoid synchronization on the session level.

Asynchronous Invocation API

Use the Asynchronous Invocation API to transfer events that require processing in a Session Initiation Protocol (SIP) application session to any server in a cluster based on an application session ID.

The Asynchronous Invocation API, also referred to as the asynchronous work dispatcher, can transfer events that require processing in the context of a SIP application session to any server in a cluster, using

the related application session ID. These transfers are usually triggered by events that cause a state change to SIP sessions that reside on another server. The asynchronous work dispatcher transfers the event task to the correct server to be run.

In Websphere Application Server, all related SIP messages are delivered to the same server in the cluster, and sessions always reside in the same SIP container. To prevent synchronicity issues and locking, tasks in the same application session cannot be processed simultaneously (that is, on different threads or processes), which limits handling of certain types of events.

For more in-depth information about SIP sessions and SIP application sessions and their relationship, see Section 6 of Java Specification Request (JSR) 289.

The following two scenarios are resolved by implementing the Asynchronous Invocation API.

1. Two requests related to the same SIP application session are simultaneously run on two different threads. For example, a Java EE (J2EE) applications working with a message-driven bean (MDB) can retrieve an event to send a SIP message on a certain SIP application session. At the same time, the SIP container receives an incoming SIP message on another session that is connected to the same application session and handles that on a different thread. It would be necessary to synchronize the session access to avoid race conditions and to ensure that all session attributes are synchronized. To employ the locking mechanism in this case would not be effective, because the SIP application session might contain multiple SIP sessions.
2. A server that does not own a certain SIP application session receives a request, through non-SIP protocol, to send a message in the context of that session. For example, a Web service that is initiating SIP dialogs can reside in a server different from the server that owns the SIP application session it must use.

The Asynchronous Invocation API ensures that the specific application code is run on the correct server and correct thread according to the SIP application session ID.

The Asynchronous Invocation API provides the following benefits:

1. No more than two servers are involved in the asynchronous invocation process: one is the server that retrieves the work task, and the other is the target server that handles the SIP application session for that task and to which the task will be transferred.
2. The asynchronous invocation allows working in a thread-safe manner. This approach ensures that only one thread processes the messages related to the SIP application session. Thus, there is no need to synchronize access to the session.
3. The asynchronous invocation provides a scalable solution. There is no performance impact when more servers are added to the cluster.
4. Cross-server invocation is used only when required, which results in better performance.

Chapter 25. Developing Spring applications

This page provides a starting point for finding information about how to develop Spring applications that can run successfully in a WebSphere Application Server environment.

The Spring Framework is an open source project that provides a framework for simple Java objects that enables them to use the Java EE container through wrapper classes and XML configuration.

Configuring access to a Spring application data source

You can use WebSphere Application Server to manage access to a data source for a Spring application.

About this task

For a Spring application to access a data source, such as a Java Database Connectivity (JDBC) data source, the application must use a resource provider that is managed by the WebSphere Application Server. For more information about Spring applications and the Spring Framework see the following topics:

- Spring Framework
- Data access and the Spring Framework

Procedure

1. During development, configure the WAR module with a resource reference. For example:

```
<resource-ref>
    <res-ref-name>jdbc/springdbres-ref-name>jdbc/springdb>
    <res-type>javax.sql.DataSourceres-type>javax.sql.DataSource>
    <res-auth>Containerres-auth>Container>
    <res-sharing-scope>Shareableres-sharing-scope>Shareable>
</resource-ref>
```

2. For Enterprise JavaBeans (EJB) Java archive (JAR) files, declare the same resource reference in each EJB that needs to access the data source. Use one of the following steps:
   - Declare a data source proxy bean. In the Spring application configuration, declare a proxy bean that references a resource provider that the application server manages. Set the value of the jndiName property to java:comp/env/ followed by the value of res-ref-name property that you declared in the resource reference. For example:

```
<bean id="wasDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/springdb"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="cache" value="true"/>
    <property name="proxyInterface" value="javax.sql.DataSource"/>
</bean>
```

   - Alternatively, for Spring Framework Version 2.5 or later, use the <j2ee:jndi-lookup/> approach. Set the value of the jndi-name property to the value of the res-ref-name property that you declared in the resource reference, and the value of the resource-ref property to `true`. For example:

```
<jee:jndi-lookup id=" wasDataSource "
    jndi-name="jdbc/springdb"
    cache="true"
    resource-ref="true"
    lookup-on-startup="false"
    proxy-interface="javax.sql.DataSource"/>
```

   The Spring application can then use the data source proxy bean as appropriate.
3. When the application is deployed to an application server, configure a resource provider and resource data source that the Spring application resource reference can use.

Results

The resource reference that is declared in the deployment descriptor of the module will be bound to the configured data source of the application server during deployment.

Chapter 26. Developing Transactions

This page provides a starting point for finding information about Java Transaction API (JTA) support. Applications running on the server can use transactions to coordinate multiple updates to resources as one unit of work, such that all or none of the updates are made permanent.

The product provides advanced transactional capabilities to help application developers avoid custom coding. It provides support for the many challenges related to integrating existing software assets with a Java EE environment.

More introduction...

Developing components to use transactions

These topics provide information about developing WebSphere application components to use transactions

About this task

The way that applications use transactions depends on the type of application component, as follows:
- A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (component-managed transactions where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) and application client components use component-managed transactions.

Use the following tasks to develop WebSphere application components that use transactions:

Procedure

- Configure transactional deployment attributes. This task determines whether EJB components use container- or bean-managed transactions by setting an appropriate value on the Transaction type deployment attribute. You can also configure other transactional deployment descriptor attributes.
- Use component-managed transactions. If you want a session bean, web component, or application client component to manage its own transactions, you must write the code that explicitly demarcates the boundaries of a transaction. There are some limitations to the transaction support available to application client components, as described in the topic about client support for transactions.

Configuring transactional deployment attributes

You can configure the transactional deployment descriptor attributes associated with an EJB or web module, to enable an enterprise application to use transactions.

Before you begin

You must have an enterprise archive (EAR) file for an application component that can be deployed in the application server.

About this task

You can configure the deployment attributes of an application by using an assembly tool.

You can use Rational Application Developer, or an equivalent tool, to configure the deployment attributes of an application.

To use Rational Application Developer to set transactional attributes in the deployment descriptor for an application component (enterprise bean or servlet), complete the following steps.

Procedure

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, use the Import wizard to import the EAR file into the assembly tool. To start the Import wizard:
   a. Click **File** > **Import** > **EAR file**.
   b. Click **Next**, then select the EAR file.
   c. Click **Finish**.
3. In the Project Explorer view of the Java EE perspective, right-click the component instance, then click **Open With** > **Deployment Descriptor Editor**. To locate the component instance, use the appropriate step:
   - For a session bean, expand **EJB Modules** > ***ejb_module_instance*** > **Deployment Descriptor** > **Session Beans**, then select the bean instance.
   - For a servlet, expand **Web Modules** > ***web_application*** > **Deployment Descriptor** > ***web component***, then select the servlet instance.

   A property dialog notebook for the deployment descriptor of the component is displayed in the property pane.
4. Optional: For session beans only, set the “Transaction type” attribute, which defines the transactional manner in which the container invokes a method. You can set this attribute to Container or Bean, as follows:
   - To use container-managed transactions, set the attribute to Container.
   - To use bean-managed transactions, set the attribute to Bean.
5. In the deployment descriptor notebook, select the **Bean** tab. Optionally, in the WebSphere Extensions section, configure the Local Transaction attributes. To enable management of local transaction containments, configure the following component extensions attributes. These attributes configure, for the component, the behavior of the local transaction containment (LTC) environment that the container establishes whenever a global transaction is not present.

   Boundary

   This setting specifies the containment boundary at which all contained resource manager local transactions (RMLTs) must be completed. Possible values are BeanMethod or ActivitySession.
   - BeanMethod: This is the default value. If you select this option, RMLTs must be resolved within the same bean method in which they were started.
   - [For EJB components only] ActivitySession: RMLTs must be resolved within the scope of any ActivitySession in which they were started or, if no ActivitySession context is present, within the same bean method in which they were started.

     Note: The ActivitySession option is not supported in the web container.

   Resolver

   This setting specifies the component responsible for initiating and ending RMLTs. Possible values are Application or ContainerAtBoundary.
   - Application: This is the default value. The application is responsible for starting RMLTs and for completing them within the local transaction containment (LTC) boundary. Any RMLTs that are not completed by the end of the LTC boundary are cleaned up by the container according to the value of the Unresolved action attribute.
   - ContainerAtBoundary: The container is responsible for starting RMLTs and for completing them within the LTC boundary. The container begins an RMLT when a connection is first used within the LTC scope, and completes it automatically at the end of the LTC scope. If Boundary is set to ActivitySession, the RMLTs are enlisted as ActivitySession resources and directed to complete by the ActivitySession. If Boundary is set to BeanMethod, the RMLTs are committed at the end of the method by the container.

   Unresolved action

Specifies the direction that the container requests RMLTs to take, if those transactions are unresolved at the end of the LTC boundary scope and the Resolver is set to Application. Possible values are Rollback or Commit.

- Rollback: This is the default value. At end of the LTC boundary scope, the container instructs all unresolved RMLTs to roll back.
- Commit: At the end of the LTC boundary scope, the container instructs all unresolved RMLTs to commit. The container instructs the RMLTs to commit only in the absence of an un-handled exception. If the application method that is running in the local transaction context ends with an exception, any unresolved RMLTs are rolled back by the container. This is the same behavior as for global transactions.

Shareable

Specifies whether the component can share an LTC. A new LTC is started only if a shareable LTC does not already exist. Applications that use shareable LTCs cannot explicitly commit or roll back resource manager connections that are used in a shareable LTC (although they can use connections that have an autoCommit capability).

If an application starts any non-autocommit work in an LTC for which the Resolver attribute is set to `Application`, and the Shareable attribute is set to `true`, an exception is thrown at run time. For example, on a JDBC Connection, non-autocommit work is work that the application performs after using the setAutoCommit(false) method to switch off the autocommit option on the connection. Enterprise beans that use bean managed transactions (BMT) cannot be assembled with the Shareable attribute set on the LTC configuration.

You must specify the **`Shareable`** attribute for all components that share the LTC. The component that creates the shareable LTC determines the other properties of the shared LTC, for example the value of the Resolver attribute.

6. In the WebSphere Extensions section, configure the Global Transaction attributes. These attributes configure, for the component, behavior in the presence of a global transaction.

Component Transaction Timeout

For enterprise beans that use container-managed transactions only, specifies the transaction timeout, in seconds, for any new global transaction that the container starts on behalf of the enterprise bean. For transactions started on behalf of the component, the Component Transaction Timeout setting overrides the default total transaction lifetime timeout that is configured in the transaction service settings for the application server.

The following attributes enable WS-AtomicTransaction and WS-BusinessActivity support for JAX-RPC applications only:

Use Web Services Atomic Transaction

For enterprise beans only, when this attribute is selected, if the application component makes any web service requests, any transaction context is propagated with the web service requests in accordance with the WebSphere WS-AtomicTransaction support described in Web Services Atomic Transaction support in the application server. When this attribute is not selected, web service requests do not carry transaction context.

Send Web Services Atomic Transaction on requests

For web components only, when this attribute is selected, if the application component makes any web service requests, any transaction context is propagated with the web service requests in accordance with the WebSphere WS-AtomicTransaction support described in Web Services Atomic Transaction support in the application server. When this attribute is not selected, web service requests do not carry transaction context.

Execute using Web Services Atomic Transaction on incoming requests

For web components only, when this attribute is selected, Web application components are prepared to run under a received WS-AtomicTransaction context. A web application component can run under a received WS-AtomicTransaction context in a similar way to an enterprise bean deployed with a container transaction type of Supports. When this attribute is not selected, the container of the web application component suspends any received

transaction context, in a similar way to the behavior of an EJB container for an enterprise bean deployed with a container transaction type of NotSupported.

If your application uses JAX-WS, enable support for WS-AtomicTransaction or WS-BusinessActivity by creating a policy set, adding the WS-Transaction policy type to the policy set, and attaching the policy set to the service or client.

If a policy set that is attached to a client includes the WS-Transaction policy type, any active global transaction context is propagated with a Web service request, in a similar way to the deployment descriptors Use Web Services Atomic Transaction and Send Web Services Atomic Transaction on requests, described earlier in this topic. Also, when the WS-Transaction policy type is included, the service runs under any received WS-AtomicTransaction context, in a similar way to the deployment descriptor Execute using Web Services Atomic Transaction on incoming requests, described earlier in this topic.

7. For EJB components only, for container-managed transactions, configure how the container manages the transaction boundaries when delegating a method invocation to the business method of an enterprise bean:
   a. In the deployment descriptor notebook, select the **Assembly** tab. The Container Transactions section displays a table of the methods for enterprise beans.
   b. For each method of the enterprise bean, set the container transaction type to an appropriate value. The default value for the container transaction type is Required, meaning that the method invocation occurs in the context of a transaction. This transaction is either the (local or remote) client component transaction or, if the client component does not run in a transaction, a new transaction started by the component container.

   If the application uses ActivitySessions, how the container manages transaction boundaries when delegating a method invocation depends on both the container transaction type that you set in this task, and the ActivitySession kind attribute, which is described in “Setting EJB module ActivitySession deployment attributes” on page 4. For more detail about the relationship between these two properties, see ActivitySession and transaction container policies in combination.
8. For web services applications that use a SOAP/JMS binding and participates in WS-AtomicTransactions, set the container transaction type of the message-driven bean named “JMS router MDB” to a value of NotSupported, as described in the previous step. Web service applications that use a SOAP/JMS binding include a router message-driven bean named “JMS router MDB” in the assembled EAR. If a web service uses a SOAP/JMS binding and participates in WS-AtomicTransactions, as described in Web Services Atomic Transaction support in the application server, set the container transaction type of the “JMS router MDB” to a value of NotSupported.

   For web services applications that use a SOAP/HTTP binding and participate in WS-AtomicTransactions, you do not have to do this.
9. For client application components only, if required, enable support for transaction demarcation by the client. In the deployment descriptor notebook, select the **Allow JTA demarcation** check box. This option directs the client container to bind the Java Transaction API (JTA) UserTransaction interface into JNDI at java:comp/UserTransaction for the client component. There are constraints on transaction support in the client container, which are described in Client support for transactions.
10. Save your changes to the deployment descriptor.
    a. Close the deployment descriptor editor.
    b. When prompted, click **Yes** to save changes to the deployment descriptor.
11. Verify the archive files. For more information about verifying files by using Rational Application Developer, refer to the Rational Application Developer information.
12. From the menu of the project, click **Deploy** to generate EJB deployment code.
13. Optional: Test your completed module on an application server installation. Right-click a module, click **Run on Server**, and follow the instructions in the resulting wizard.

Important: Use the **Run On Server** option for unit testing only. The assembly tool controls the application server installation, and when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool, for example the administrative console, to deploy the EAR file onto the application server that is to run the application.

Using component-managed transactions

You can enable a session bean, servlet, or application client component to use component-managed transactions, to manage its own transactions directly instead of letting the container manage the transactions.

About this task

Note: Entity beans cannot manage transactions (so cannot use bean-managed transactions).

To enable a session bean, servlet, or application client component to use component-managed transactions, complete the following steps:

Procedure

1. For session beans, set the **`Transaction type`** attribute in the component deployment descriptor to `Bean`, as described in “Configuring transactional deployment attributes” on page 1063.
2. For application client components, enable support for transaction demarcation by setting the **`Allow JTA Demarcation`** attribute in the component deployment descriptor, as described in “Configuring transactional deployment attributes” on page 1063.
3. Write the component code to actively manage transactions.

   For stateful session beans, a transaction started in a given method does not have to be completed (that is, committed or rolled back) before completing that method. The transaction can be completed at a later time, for example on a subsequent call to the same method, or even within a different method. However, it is usually preferable to construct the application so that a transaction is begun and completed within the same method call, because it simplifies application debugging and maintenance.

   The following code extract shows the standard code required to obtain an object encapsulating the transaction context, and involves the following steps:
   - Create a javax.transaction.UserTransaction object by calling a lookup on java:comp/UserTransaction.
   - Use the UserTransaction object to demarcate the boundary of a transaction by using transaction methods such as begin and commit, as needed. If an application component begins a transaction, it must also complete that transaction by invoking either the commit method or the rollback method.

```
...
import javax.transaction.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
...
    public float doSomething(long arg1)throws NamingException {
        InitialContext initCtx = new InitialContext();
        UserTransaction userTran = (UserTransaction)initCtx.lookup(
           "java:comp/UserTransaction");
        ...
        //Use userTran object to call transaction methods
        userTran.begin ();
        //Do transactional work
        ...
        userTran.commit ();
```

```
        ...
    }
    ...
}
```

Using one-phase and two-phase commit resources in the same transaction

Use these topics to help you coordinate the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction.

About this task

You can coordinate the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction. You can have multiple interactions that involve the one-phase commit resource in the same transaction, but only one such resource can be involved. This coordination is enabled by the *last participant support*.

At transaction commit, the two-phase commit resources are prepared first using the two-phase commit protocol, and if this is successful the one-phase commit-resource is then called to commit. The two-phase commit resources are then committed or rolled back depending on the response of the one-phase commit resource.

For more information about using one-phase and two-phase commit resources within the same transaction, see the following topics:

Procedure

- “Coordination of access to one-phase commit and two-phase commit capable resources in the same transaction” on page 1070
- “Assembling an application to use one-phase and two-phase commit resources in the same transaction”
- Configuring an application server to log heuristic reporting

Assembling an application to use one-phase and two-phase commit resources in the same transaction

Use this task to assemble an application to use one-phase and two-phase commit resources in the same transaction.

Before you begin

This task description assumes that you have an EAR file for an application component that can be deployed in WebSphere Application Server. For more details about assembling applications, see the topic about assembling applications.

About this task

To enable an application to use one-phase and two-phase commit capable resources in the same transaction, you must configure the deployment attributes of the application to accept the heuristic hazard, that is, the increased risk of an heuristic outcome. You can configure the deployment attributes of an application by using an assembly tool.

You can also configure an application to accept the heuristic hazard after deployment, by using the administrative console and the Last participant support extension settings. Alternatively, you can configure the transaction service for an application server to accept the heuristic hazard.

This topic describes the use of Rational Application Developer to configure the deployment attributes of an application.

To configure an application to indicate that you accept the increased risk of an heuristic outcome, complete the following steps:

Procedure

1. Start the assembly tool. For more information, refer to the Rational Application Developer information.
2. Create or edit the application EAR file.

   Note: Ensure that you set the target server as WebSphere Application Server Version 7.0.

   For example, to change attributes of an existing application, use the Import wizard to import the EAR file into the assembly tool. To start the Import wizard:

   a. Click **File** > **Import** > **EAR file**.
   b. Click **Next**, then select the EAR file.
   c. In the Target server field, select `WebSphere Application Server v7.0`.
   d. Click **Finish**.
3. In the Project Explorer view of the Java EE perspective, complete the following steps:
   a. Expand the Enterprise Application instance.
   b. Right click on the Deployment Descriptor.
   c. Click **Open With** > **Deployment Descriptor Editor**.

   A property dialog notebook for the component is displayed in the property pane.
4. Complete the following steps to display the Extended Services tab.
   a. Close the Enterprise Application Deployment Descriptor editor.
   b. In the toolbar, select **Windows** > **Preferences**.
   c. In the left pane, select **Capabilities**.
   d. In the right pane, expand **Advanced Java EE** and select the **WebSphere PME Development** option.
   e. Click **Apply**.
   f. Open the Enterprise Application Deployment Descriptor editor.
5. On the Extended Services tab, in the Last Participant Support section, select the **Last participant support** check box.
6. Save your changes to the deployment descriptor.
   a. Close the Deployment Descriptor Editor.
   b. When prompted, click **Yes** to save changes to the deployment descriptor.
7. Verify the archive files. For more information about verifying files using Rational Application Developer, refer to the Rational Application Developer information.
8. From the popup menu of the project, click **Deploy** to generate EJB deployment code.
9. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard.

   Important: Use **Run On Server** only for unit testing. The assembly tool controls the WebSphere Application Server installation and, when an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use the **Run On Server** option on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console, as described in the topic about deploying and administering enterprise applications.

Last participant support extension settings:

Use this page to configure settings for last participant support. Last participant support is an extension to the transaction service that enables a single one-phase resource to participate in a two-phase transaction with one or more two-phase resources. Values on this panel are ignored if you select **Use configuration information in binary** on the Application binaries panel.

To view this administrative console page, click **Applications > Application Types > WebSphere enterprise applications >** *application_name*. Under **Detailed Properties**, click **Last participant support extension**.

Accept heuristic hazard:

Specifies whether an application accepts the possibility of a heuristic hazard occurring in a two-phase transaction that contains a one-phase resource.

Information	**Value**
Default	Cleared
Range	**Selected** The application accepts the increased risk of an heuristic outcome. **Cleared** The application does not accept the increased risk of an heuristic outcome.

Coordination of access to one-phase commit and two-phase commit capable resources in the same transaction

Last participant support enables the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction. You can have multiple interactions that involve the one-phase commit resource in the same transaction, but only one such resource can be involved.

At transaction commit, the two-phase commit resources are prepared first using the two-phase commit protocol, and if this is successful, the one-phase commit-resource is then called to commit. The two-phase commit resources are then committed or rolled back, depending on the response of the one-phase commit resource.

Note: If the global transaction is distributed across multiple application servers *that are all running at WebSphere Application Server version 6.1 or later*, you can exploit last participant support to coordinate a one-phase commit capable resource and any number of two-phase commit capable resources in the same transaction, in a limited number of scenarios.

- The main scenario is where the one-phase commit resource provider is accessed in the application server process (the “transaction root” server) in which the transaction is started.

  In this scenario, last participant support can coordinate a one-phase commit capable resource and any number of two-phase commit capable resources in the same transaction.
- If the one-phase commit resource provider is accessed in a different application server (a “transaction subordinate” server) from the one in which the transaction was started; for example, as a result of a transactional invocation on a remote EJB interface where the EJB implementation accesses a one-phase commit resource provider.

  In this scenario, the transaction typically cannot be committed. To be able to commit (as part of a global transaction) a one-phase commit resource enlisted on a transaction subordinate server, the transaction service must delegate coordination responsibility from the transaction root to the subordinate server. This occurs only if no other resources were registered with the transaction root server.

Last participant support introduces an increased risk of an heuristic outcome to the transaction. That is, the transaction manager cannot be sure that all resources were completed in the same direction (either committed or rolled back). For this reason, to enable an application to coordinate access to one-phase and two-phase commit capable resources in the same transaction, you configure the application to accept the heuristic hazard, that is, accept the increased risk of an heuristic outcome.

An heuristic outcome occurs if the transaction service (JTS) receives no response from the commit one-phase flow on the one-phase commit resource. In this situation, the transaction service cannot determine whether changes for the one-phase commit resource were committed or rolled back, so cannot drive reliably the correct outcome of the global transaction on the other two-phase commit resources.

You can configure the transaction service for an application server to accept the heuristic hazard, or you can configure applications individually to accept the heuristic hazard. You can configure applications individually either when they are assembled, or after they are deployed.

You can configure the transaction service for an application server to indicate whether or not to log that it is about to commit the one-phase commit resource. This does not reduce the heuristic hazard, but ensures that any failure, and subsequent recovery, of the application server during the one-phase commit phase occurs with knowledge of whether or not the one-phase commit resource was asked to commit:
- If the one-phase commit resource was asked to commit, a heuristic outcome is reported to the activity log.
- If the one-phase commit resource was not asked to commit, then the transaction is rolled back consistently.

Transaction exceptions that involve both one-phase and two-phase commit resources

The exceptions that can be thrown by transactions that involve one-phase and two-phase commit resources are the same as those that can be thrown by transactions involving only two-phase commit resources.

The exceptions that can occur are listed in the application programming interface (API) reference information in the WebSphere Application Server information center.

Last Participant Support: Resources for learning:

Use the links in this topic to find relevant supplemental information about Last Participant Support. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming specifications
- J2EE Activity Service for Extended Transactions
- Java Transaction API (JTA) 1.0.1

Other
- WebSphere Business Integration Server Foundation
- List of IBM WebSphere Redbooks
- WebSphere technical library, including links to white papers

Chapter 27. Developing web applications

This page provides a starting point for finding information about web applications, which are comprised of one or more related files that you can manage as a unit, including:

- HTML files
- Servlets can support dynamic web page content, provide database access, serve multiple clients at one time, and filter data.
- Java ServerPages (JSP) files enable the separation of the HTML code from the business logic in web pages.

IBM extensions to the JSP specification make it easy for HTML authors to add the power of Java technology to web pages, without being experts in Java programming. More introduction...

Developing web applications

Learn about selecting tools for developing web applications.

Before you begin

Design a web application and the required components.

About this task

There are two basic approaches to selecting tools for developing web applications:

- You can use one of the available integrated development environments (IDEs). IDE tools automatically generate significant parts of the servlet and JavaServer Pages (JSP) code, and Hypertext Markup Language (HTML) files. They also contain integrated tools for packaging and testing the web application components.
- If you decide to develop web components without an IDE, you need at least an ASCII text editor. You can also use tools available in the Java SE Development Kit 6 and in this product to assemble, test, and deploy the Web application components.

The following steps support the second approach, development without an IDE.

Procedure

1. If necessary, migrate any pre-existing code to the required version of the servlet and JSP specification.
2. Write and compile the components of the web application. To access classes that were extended, compile your code using the `-classpath` option on the `javac` compiler. This option allows you to reference the `j2ee.jar` file in the product directory:
   - *<install_root>*`\dev\JavaEE`

   To compile that same servlet on the Windows NT version of WebSphere Network Deployment, specify:

   ```
   javac -classpath D:\Program Files\WebSphere\DeploymentManager\dev\JavaEE\j2ee.jar MyServlet.java
   ```
3. Optionally disable JavaServer Pages (JSP) runtime compilation, if necessary.

What to do next

Assemble the application components in one or more web modules.

Developing servlets

Developing servlets with WebSphere Application Server extensions

Use this task to provide a summary of the WebSphere Application Server extensions that you can use to develop servlets.

About this task

Several WebSphere Application Server extensions are provided for enhancing your servlets. This task provides a summary of the extensions that you can utilize.

Procedure

1. Review the supported specifications.

   Create Java components, referring to the Servlet specifications.

   The application server includes its own packages that extend and add to the Java Servlet Application Programming Interface (API). These extensions and additions make it easier to manage session states, create personalized Web pages, generate better servlet error reports, and access databases. Locate the API documentation for the application server APIs in the `install_root\web\apidocs` directory for the default installation. All of the public Application Server APIs are located in the `com.ibm.websphere` packages, however, com.ibm.websphere.servlet package is specific to the product servlet APIs.
2. Use your favorite integrated development environment (IDE), or a text editor, to develop or migrate code artifacts that meet the specifications.
3. Test the code artifacts.

What to do next

Assemble your code artifacts into a web module using assembly tools as a prerequisite to deploying the code to the application server.

Configuring page list servlet client configurations:

You can define PageListServlet configuration information in the IBM Web Extensions file. The IBM Web Extensions file is created and stored in the web applications archive (WAR) file by an assembly tool.

About this task

Attention: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 8.5 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes. Starting from the Servlet 2.3 specification, javax.servlet.filter classes you can intercept requests and examine responses. You can also use javax.servlet.filter classes to achieve chaining functionality, as well as embellishing or truncating responses.

To configure and implement page lists:

Procedure

1. To configure page list information, use the Add Markup Language entry dialog of an assembly tool. On the Servlets tab of a web deployment descriptor editor, select a servlet and click **Add** under WebSphere Extensions.
2. Add the `callPage()` method to your servlet to invoke a JavaServer Page (JSP) file in response to a client request.

   The PageListServlet has a `callPage()` method that invokes a JSP file in response to the HTTP request for a page in a page list. The `callPage()` method can be invoked in one of the following ways:

- `callPage(String pageName, HttpServletRequest request, HttpServletResponse response)`

  where the method arguments are:

  pageName
  A page name defined in the PageListServlet configuration

  request
  The HttpServletRequest object

  response
  The HttpServletResponse object
- `callPage(String mlName, String pageName, HttpServletRequest request, HttpServletResponse response)`

  where the method arguments are:

  mlName A markup language type

  pageName
  A page name defined in the PageListServlet configuration

  request
  The HttpServletRequest object

  response
  The HttpServletResponse object

3. Use the PageList Servlet client type detection support to determine the markup language type a calling client requires for the response.

Extending PageListServlet

The following example shows how a servlet extends the PageListServlet class and determines the markup-language type required by the client. The servlet then uses the callPage method to call an appropriate JavaServer Pages (JSP) file. In this example, the JSP file that provides the correct markup-language for the response is *Hello.page*.

```
public class HelloPervasiveServlet extends PageListServlet implements Serializable
{
     /*
     * doGet -- Process incoming HTTP GET requests
     */
    public  void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
       // This is the name of the page to be called:
       String pageName = "Hello.page";

       // First check if the servlet was invoked with a queryString that contains
    // a markup-language value.
       // For example, if this is how the  servlet is invoked:
       //  http://localhost/servlets/HeloPervasive?mlname=VXML
       //  then use the following method:
       String mlname= getMLNameFromRequest(request);

        // If no markup language type is provided in the queryString,
        // then try to determine the client
        // Type from the request, and use the markup-language name configured in
        // the client_types.xml file.
        if (mlName == null)
        {
          mlName = getMLTypeFromRequest(request);
         }
         try
         {
           // Serve the request page.
           callPage(mlName, pageName, request, response);
          }
          catch (Exception e)
          {
```

```
            handleError(mlName, request, response, e);
           }
       }
}
```

Page lists:

Page lists allow you to avoid hard-coding Uniform Resource Locators (URLs) in servlets and JSP files. A page list specifies the location where a request is to be forwarded, but automatically customizes that location depending on the MIME type of the servlet. Use these properties to specify a markup language and an associated MIME type. For the given MIME type, you also specify a set of pages to invoke.

Note: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 8.5 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes. Starting from the Servlet 2.3 specification, javax.servlet.filter classes you can intercept requests and examine responses. You can also use javax.servlet.filter classes to achieve chaining functionality, as well as embellishing or truncating responses.

The following list of classes are deprecated:

- com.ibm.servlet.ClientList
- com.ibm.servlet.ClientListElement
- com.ibm.servlet.MLNotFoundException
- com.ibm.servlet.PageListServlet
- com.ibm.servlet.PageNotFoundException

WebSphere Application Server supplies the PageListServlet servlet, which you can use to call a JavaServer Pages (JSP) file by name based on the configuration data in the `client_types.xml` file. This file maps a JSP file to a Uniform Resource Identifier (URI). When the URI is invoked, it specifies another JSP file in a web module. This support allows you to access multiple URLs without hard-coding them in your servlets.

You can also logically group page lists according to the markup language type, such as, Hypertext Markup Language (HTML) or Wireless Markup Language (WML). This allows applications that use servlets to extend the PageListServlet servlet, to call JSP files which return the proper markup-language type for the client request. For example, a request that originates from a PDA device requires WML data. The application server sends the request to a servlet that extends the PageListServlet servlet, and the servlet calls a JSP file that returns a WML response.

Client type detection support:

In addition to providing the page list mapping capability, the PageListServlet also provides Client Type Detection support. A servlet determines the markup language type that a calling client needs in the response, using the configuration information in the `client_types.xml` file.

Client type detection support allows a servlet, extending the PageListServlet, to call an appropriate JavaServer Pages (JSP) file. The servlet invokes the **`callPage`** method, which calls a JSP file based on the markup-language type of the request.

The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 8.5 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes.

The client_types.xml file:

The client_types.xml file provides client type detection support for servlets extending PageListServlet. Using the configuration data in the client_types.xml file, servlets can determine the language type that calling clients require for the response.

Attention: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 8.5 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes.

The client type detection support allows servlets to call appropriate JavaServer Pages (JSP) files with the callPage method. Servlets select JSP files based on the markup-language type of the request.

Servlets must use the following version of the callPage method to determine the markup language type required by the client:

```
callPage(String mlName, String pageName, HttpServletRequest request,
                  HttpServletResponse response)
```

where the arguments are:
- mlName - a markup language type
- pageName - a page name defined in the PageListServlet configuration
- request - the HttpServletRequest object
- response - the HttpServletResponse object

Review the Extending the PageListServlet code example in the Extending the PageListServlet topic, to see how the callPage method is invoked by a servlet.

In the example, the client type detection method, getMLTypeFromRequest(HttpServletRequestrequest), provided by the PageListServlet, inspects the HttpServletRequest object request headers, and searches for a match in the client_types.xml file.

The client type detection method does the following:
- Uses the input HttpServletRequest and the client_types.xml file, to check for a matching HTTP request name and value.
- Returns the markup-language value configured for the <client-type> element, if a match is found.
- If multiple matches are found, this method returns the markup-language for the first <client-type> element for which a match is found.
- If no match is found, returns the value of the markup-language for the default page defined in the PageListServlet configuration.

Location

The client_types.xml file is located in the *install_root*/properties directory.

Usage notes

Review the answers to the following usage questions:
- Is this file read-only?

  No
- Is this file updated by a product component?

  No
- If so, what triggers its update?

  This file is created and updated manually by users.
- How and when are the contents of this file used?

  Servlets that extending the PageListServlet servlet use this file to determine the language type that calling clients require for the response.

Sample file entry

```
<?xml version="1.0" >
<!DOCTYPE clients [
<!ELEMENT client-type (description, markup-language,request-header+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT markup-language (#PCDATA)>
<!ELEMENT request-header (name, value)>
<!ELEMENT clients (client-type+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>]>
<clients>
      <client-type>
      <description>IBM Speech Client</description>
      <markup-language>VXML</markup-language>
      <request-header>
           <name>user-agent</name>
           <value>IBM VoiceXML pre-release version 000303</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vxml</value>
      </request-header>
      </client-type>
      <client-type>
           <description>WML Browser</description>
           <markup-language>WML</markup-language>
      <request-header>
           <name>accept</name>
           <value>text/x-wap.wml</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vnd.wap.xml</value>
      </request-header>
    </client-type>
</clients>
```

The client_types.xml file:

The `client_types.xml` file provides client type detection support for servlets extending PageListServlet. Using the configuration data in the `client_types.xml` file, servlets can determine the language type that calling clients require for the response.

Attention: The PageList Servlet custom extension is deprecated in WebSphere Application Server Version 8.5 and will be removed in a future release. Re-architect your legacy applications to use javax.servlet.filter classes instead of com.ibm.servlet classes.

The client type detection support allows servlets to call appropriate JavaServer Pages (JSP) files with the `callPage` method. Servlets select JSP files based on the markup-language type of the request.

Servlets must use the following version of the `callPage` method to determine the markup language type required by the client:

```
callPage(String mlName, String pageName, HttpServletRequest request,
                  HttpServletResponse response)
```

where the arguments are:
- mlName - a markup language type
- pageName - a page name defined in the PageListServlet configuration
- request - the HttpServletRequest object

- response - the HttpServletResponse object

Review the Extending the PageListServlet code example in the Extending the PageListServlet topic, to see how the `callPage` method is invoked by a servlet.

In the example, the client type detection method, `getMLTypeFromRequest(HttpServletRequestrequest)`, provided by the PageListServlet, inspects the HttpServletRequest object request headers, and searches for a match in the `client_types.xml` file.

The client type detection method does the following:
- Uses the input HttpServletRequest and the `client_types.xml` file, to check for a matching HTTP request name and value.
- Returns the markup-language value configured for the `<client-type>` element, if a match is found.
- If multiple matches are found, this method returns the markup-language for the first `<client-type>` element for which a match is found.
- If no match is found, returns the value of the markup-language for the default page defined in the PageListServlet configuration.

Location

The `client_types.xml` file is located in the *`install_root`*`/properties` directory.

Usage notes

Review the answers to the following usage questions:
- Is this file read-only?

  No
- Is this file updated by a product component?

  No
- If so, what triggers its update?

  This file is created and updated manually by users.
- How and when are the contents of this file used?

  Servlets that extending the PageListServlet servlet use this file to determine the language type that calling clients require for the response.

Sample file entry

```
<?xml version="1.0" >
<!DOCTYPE clients [
<!ELEMENT client-type (description, markup-language,request-header+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT markup-language (#PCDATA)>
<!ELEMENT request-header (name, value)>
<!ELEMENT clients (client-type+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>]>
<clients>
      <client-type>
      <description>IBM Speech Client</description>
      <markup-language>VXML</markup-language>
      <request-header>
           <name>user-agent</name>
           <value>IBM VoiceXML pre-release version 000303</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vxml</value>
      </request-header>
```

```
      </client-type>
      <client-type>
           <description>WML Browser</description>
           <markup-language>WML</markup-language>
      <request-header>
           <name>accept</name>
           <value>text/x-wap.wml</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vnd.wap.xml</value>
      </request-header>
    </client-type>
</clients>
```

Java Servlet 3.0 considerations:

When using Servlet 3.0 web modules, keep in mind the following features.

Java Servlet 3.0 has many, new powerful features. Some of these features are not fully documented in the Servlet 3.0 specification or they entail trade-offs. Consider the following topics to make best use of the new features.

Annotations

Java Servlet 3.0 annotations are picked up in Servlet 2.5 web modules, which can include exposing a servlet on the web. Use caution when upgrading prerequisites of an older application, because the new annotations are processed and the prerequisites JAR file might include annotations that you do not want applied.

File upload

When using the file upload (multipart forms) support that is new to Servlet 3.0, the default location for writing files is the same as the value of the javax.servlet.context.tempdir servlet context attribute. For example, `C:\opt\WAS\profiles\node1\temp\node1\server1\fragmentTest\fragmentTest24.war` is produced for a configuration with the following attributes:

- profile home=C:\opt\WAS\profiles\node1
- server name=server1
- enterprise application name=fragmentTest
- web module name=fragmentTest24.war

Relative paths are also relative to this default location.

You can change the value of the javax.servlet.context.tempdir servlet context attribute to be relative to a different directory by setting the com.ibm.websphere.servlet.temp.dir system property. This system property affects all applications on a server-wide basis. For example, if you set com.ibm.websphere.servlet.temp.dir to `/foo`, the application temp directory is `/foo/node1/server1/fragmentTest/fragmentTest24.war`. If you want to change the value at an application level, use the scratchdir JavaServer Pages (JSP) attribute. View the JSP engine configuration parameters topic for more information about the scratchdir attribute.

Programmatic or dynamic HTTP session configuration

Programmatic HTTP session configuration enables an application to modify the session configuration in use, either through `web.xml` file configuration or through API method calls. After the application starts, a dynamically modified cookie name cannot be changed. For security purposes, administrators can disable programmatic session configuration for particular cookies that can be shared between applications.

Generally, it is safe to modify the cookie configuration, if the application uses a unique cookie name or path. You can change the default cookie path for each application to use the context root through the session management.

Important: Changing the path can affect certain IBM extensions, such as session sharing or the IBMSessionExt.invalidateAll method that rely on using one cookie for multiple applications.

Dynamic cookies have the following impact on intermediary services:

- An enterprise proxy automatically retrieves a dynamic cookie when an application starts and uses the cookie for session affinity.
- A DMZ proxy in low or medium secure mode also automatically retrieves a dynamic cookie when an application starts. For a DMZ proxy in high secure mode, the retrieval is not automatic; the application must be running before the target routing information is exported.
- A web server plug-in cannot obtain the dynamic cookie automatically because it does not communicate with application servers for configuration information. You must start the application, generate the plug-in configuration, propagate the configuration to the plug-in, and then reload the configuration for the plug-in to obtain the cookie.
- In the cluster environment, the generated dynamic cookie name on each server must be the same, otherwise the front-end intermediary services might not be able to route requests.

Servlet 3.0 programmatic configuration:

The configuration methods, addListener, addFilter, and addServlet are introduced in the Servlet 3.0 specification.

The methods for Servlet 3.0 are part of the ServletContext interface. You can call these methods from either a ServletContainerInitializer or a ServletContextListener.

addListener

The addGlobalListener method is deprecated in WebSphere Application Server Version 8.5. It is replaced with the addListener method.

- Use the following method to add the listener with the given class name to this servlet context:
  ```
  void addListener(java.lang.String className)
  ```
- Use the following method to add the given listener to this servlet context:
  ```
  <T extends java.util.EventListener> void addListener(T t)
  ```
- Use the following method to add a listener of the given class type to this servlet context:
  ```
  void addListener(java.lang.Class<? extends java.util.EventListener> listenerClass)
  ```

The given listener class must implement one or more of the following interfaces:

- ServletContextAttributeListener
- ServletRequestListener
- ServletRequestAttributeListener
- HttpSessionListener
- HttpSessionAttributeListener

addFilter

The addMappingFilter method is deprecated in WebSphere Application Server Version 8.5. It is replaced with the addFilter method. This method adds the filter with the given name, description, and class name to the Web application context. The registered filter might be further configured using the returned FilterRegistration object.

- Use the following method to add the filter with the given name and class type to this servlet context:
  `addFilter(java.lang.String filterName, java.lang.Class<? extends Filter> filterClass)`
- Use the following method to register the given filter instance with this servlet context under the given filterName:
  `addFilter(java.lang.String filterName, Filter filter)`
- Use the following method to add the filter with the given name and class name to this servlet context:
  `addFilter(java.lang.String filterName, java.lang.String className)`

addServlet

The addServlet methods dynamically adds servlets to a servletContext. These methods will add the servlet with the given parameters to the web application context. The registered servlet might be further configured using the returned ServletRegistration object.

- Use the following method to add the filter with the given name and class type to this servlet context:
  `addFilter(java.lang.String filterName, java.lang.Class<? extends Filter> filterClass)`
- Use the following method to register the given filter instance with this servlet context under the given filterName:
  `addFilter(java.lang.String filterName, Filter filter)`
- Use the following method to add the filter with the given name and class name to this servlet context:
  `addFilter(java.lang.String filterName, java.lang.String className)`

ServletContainerInitializer

When you configure a JAR file for a shared library and a ServletContainerInitializer is discovered within the JAR, the ServletContainerInitializer is invoked for every application that the shared library associates with.

Deprecated classes in Servlet 3.0

The following classes are deprecated from com.ibm.websphere.servlet.context.IBMServletContext:

- public void addDynamicServlet(String servletName, String servletClass, String mappingURI, Properties initParameters) throws ServletException, java.lang.SecurityException;
- public void removeDynamicServlet(String servletName) throws java.lang.SecurityException

There is no replacement for the removeDynamicServlet method because removing a servlet can lead to timing issues if a request was servicing that servlet at the same time. The addServlet and createServlet methods replace the addDynamicServlet method.

Initial parameters for servlets settings:

Use this page to specify initial parameters that are passed to the init method of web module servlet filters. You can specify initial parameter values for servlets in web modules during or after installation of an application onto a WebSphere Application Server deployment target. The `<param-value>` values specified in `<init-param>` statements in the `web.xml` file of web modules are used by default.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Init parameters for servlets**. This page is the same as the Init parameters for servlets in each web module panel on the application installation and update wizards.

Module:

Specifies the name of a module in the application that you are installing or that you are viewing after installation.

URI:

Specifies the location of the module relative to the root of the application (EAR file).

Servlet:

Specifies a unique name for the servlet within the application.

A *servlet* is a Java program that uses the Java Servlet Application Programming Interface (API). You must package servlets in a Web archive (WAR) file or web module for deployment to an application server. Servlets run on a Java-enabled web server and extend the capabilities of a web server, similar to the way applets run on a browser and extend the capabilities of a browser.

Name:

Specifies the name of the initial parameter passed to the init method of the web module servlet filter.

The following example servlet filter statement in a `web.xml` file specifies an initial parameter name of `attribute`:

```
<init-param>
   <param-name>attribute</param-name>
   <param-value>tests.Filter.DoFilter_Filter.SERVLET_MAPPED</param-value>
</init-param>
```

Value:

Specifies the value assigned to an initial parameter passed to the init method of the web module servlet filter.

The following example servlet filter statement in a `web.xml` file specifies an initial parameter value of `tests.Filter.DoFilter_Filter.SERVLET_MAPPED` for the init parameter `attribute`:

```
<init-param>
   <param-name>attribute</param-name>
   <param-value>tests.Filter.DoFilter_Filter.SERVLET_MAPPED</param-value>
</init-param>
```

Description:

Specifies information on the initial parameter.

Servlet filtering:

Servlet filtering provides a new type of object called a filter that can transform a request or modify a response.

You can chain filters together so that a group of filters can act on the input and output of a specified resource or group of resources.

Filters typically include logging filters, image conversion filters, encryption filters, and Multipurpose Internet Mail Extensions (MIME) type filters (functionally equivalent to the servlet chaining). Although filters are not servlets, their life cycle is very similar.

Filters are handled in the following manner:

1. The web container determines whether it needs to construct a `FilterChain` containing the `LoggingFilter` for the requested resource.

   The `FilterChain` begins with the invocation of the `LoggingFilter` and ends with the invocation of the requested resource.

2. If other filters need to go in the chain, the web container places them after `theLoggingFilter` and before the requested resource.
3. The web container then instantiates and initializes the `LoggingFilter` (if it was not done previously) and invokes its `doFilter(FilterConfig)` method to start the chain.
4. The `LoggingFilter` preprocesses the request and response objects and then invokes the filter chain `doFilter(ServletRequest, ServletResponse)` method.

   This method passes the processing to the next resource in the chain, the requested resource.
5. Upon return from the filter chain `doFilter(ServletRequest, ServletResponse)` method, the `LoggingFilter` performs post-processing on the request and response object before sending the response back to the client.

Java Servlet Specification 2.4 enables you to define a new `<dispatcher>` element in the deployment descriptor with possible values such as REQUEST, FORWARD, INCLUDE, ERROR, instead of invoking filters with RequestDispatcher.

Java Servlet Specification 3.0 enables you to define a new `<dispatcher>` element in the deployment descriptor with possible values such as ASYNC, REQUEST, FORWARD, INCLUDE, ERROR, instead of invoking filters with RequestDispatcher.

For example:

```
<filter-mapping>
<filter-name>Logging Filter</filter-name>
<url-pattern>/products/*</url-pattern>
<dispatcher>FORWARD</dispatcher>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

This indicates that the filter should be applied to requests directly from the client as well as forward requests. Adding the INCLUDE and ERROR values also indicates that the filter should additionally be applied for included requests and `<error-page>` requests. If you do not specify any <dispatcher> elements, then the default is REQUEST.

Filter, FilterChain, FilterConfig classes for servlet filtering

The following interfaces are defined as part of the javax.servlet package:
- Filter interface - methods: doFilter, getFilterConfig, setFilterConfig
- FilterChain interface - methods: doFilter
- FilterConfig interface - methods: getFilterName, getInitParameter, getInitParameterNames, getServletContext

The following classes are defined as part of the javax.servlet.http package:
- HttpServletRequestWrapper - methods: See the Servlet 2.4 Specification
- HttpServletResponseWrapper - methods: See the Servlet 2.4 Specification

autoRequestEncoding and autoResponseEncoding:

Starting with WebSphere Application Server Version 5, the web container no longer automatically sets request and response encodings, and response content types. Programmers are expected to set these values using available methods in the Servlet 2.3 specification or later. If programmers choose not to use the character encoding methods, they can specify the **autoRequestEncoding** and **autoResponseEncoding** extensions, which enable the application server to set the encoding values and content type.

The values of the **autoRequestEncoding** and **autoResponseEncoding** extensions are either `true` or `false`. The default value for both extensions is `false`. If the value is `false` for both **autoRequestEncoding** and **autoResponseEncoding**, then the request and response character encoding is set to the Servlet Specification default, which is ISO-8859-1. Also, if the value is set to `false` for a response, the web

container cannot set a response content type. Different character encodings are possible if the client defines character encoding in the request header, or if the code includes the **setCharacterEncoding(String encoding)** method.

If the **autoRequestEncoding** value is set to `true`, and the client did not specify character encoding in the request header, and the code does not include the **setCharacterEncoding(String encoding)** method, the web container tries to determine the correct character encoding for the request parameters and data.

Use an assembly tool to change the default values for the **autoRequestEncoding** and **autoResponseEncoding** extensions.

The web container performs each step in the following list until a match is found:
- Looks at the character set (charset) in the **Content-Type** header.
- Attempts to map the servers locale to a character set using defined properties.
- Attempts to use the **DEFAULT_CLIENT_ENCODING** system property, if one is set.
- Uses the ISO-8859-1 character encoding as the default.

If the **autoResponseEncoding** value is set to `true`, and the client did not specify character encoding in the request header, and the code does not include the **setCharacterEncoding(String encoding)** method, the web container does the following:
- Attempts to determine the response content type and character encoding from information in the request header.
- Uses the ISO-8859-1 character encoding as the default.

Application life cycle listeners and events:

With application life cycle listeners and events, which are now part of the Servlet API, you can notify interested listeners when servlet contexts and sessions change. For example, you can notify users when attributes change and if sessions or servlet contexts are created or destroyed.

The life cycle listeners give the application developer greater control over interactions with ServletContext and HttpSession objects. Servlet context listeners manage resources at an application level. Session listeners manage resources that are associated with a series of requests from a single client. Listeners are available for life cycle events and for attribute modification events. The listener developer creates a class that implements the javax listener interface, corresponding to the listener functionality that you want.

At application startup time, the container uses introspection to create an instance of your listener class and registers it with the appropriate event generator.

When a servlet context is created, the contextInitialized method of your listener class is invoked, which creates the database connection for the servlets in your application to use if this context is for your application. All servlet context listeners are notified of context initialization before any servlet in the web application is initialized.

When the servlet context is destroyed, your contextDestroyed method is invoked, which releases the database connection, if this context is for your application. You must destroy all servlets before any servlet context listeners are notified of context destruction.

Notifications to session listeners precede notifications to context listeners.

Listener classes for servlet context and session changes

The following methods are defined as part of the javax.servlet.ServletContextListener interface:
- void contextInitialized(ServletContextEvent)

Notification that the web application is ready to process requests. Place code in this method to see if the created context is for your web application and if it is, allocate a database connection and store the connection in the servlet context.

- void contextDestroyed(ServletContextEvent)

  Notification that the servlet context is about to shut down. Place code in this method to see if the created context is for your web application and if it is, close the database connection stored in the servlet context.

The following methods are defined as part of the `javax.servlet.ServletRequestListener` interface:

- public void requestInitialized(ServletRequestEvent re)
  - Notification that the request is about to come into scope

    A request is defined as coming into scope when it is about to enter the first filter in the filter chain that processes the request.
- public void requestDestroyed(ServletRequestEvent re)
  - Notification that the request is about to go out of scope

    A request is defined as going out of scope when it exits the last filter in its filter chain.

The following listener interfaces are defined as part of the javax.servlet package:

- ServletContextListener
- ServletContextAttributeListener

The following filter interface is defined as part of the javax.servlet package:

- FilterChain interface - methods: doFilter()

The following event classes are defined as part of the javax.servlet package:

- ServletContextEvent
- ServletContextAttributeEvent

The following interfaces are defined as part of the javax.servlet.http package:

- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionActivationListener

The following event class is defined as part of the javax.servlet.http package:

- HttpSessionEvent

Example: Creating a servlet context listener with com.ibm.websphere.DBConnectionListener.java

The following example shows how to create a servlet context listener:

```
package com.ibm.websphere;

import java.io.*;
import javax.servlet.*;

public class DBConnectionListener implements ServletContextListener
{
     // implement the required context init method
    void contextInitialized(ServletContextEvent sce)
    {
    }

     // implement the required context destroy method
    void contextDestroyed(ServletContextEvent sce)
    {
    }
}
```

Developing JSP files

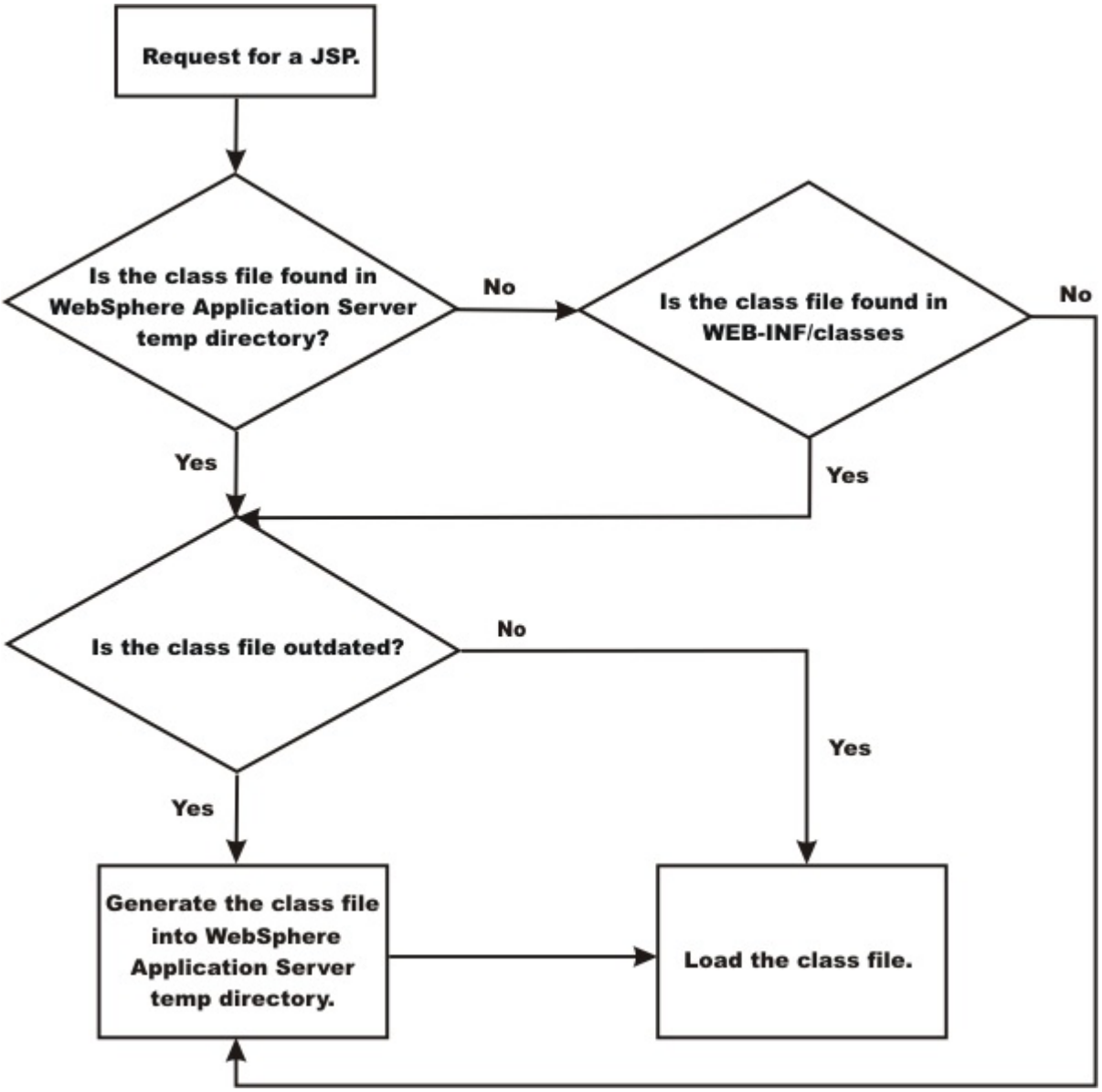
Learn about JSP files.

JSP class file generation

At runtime, the WebSphere Application Server JavaServer Pages (JSP) engine loads JSP class files from either the WebSphere Application Server `temp` directory or a web module's `WEB-INF/classes` directory. The JSP engine first searches for a class file in the `temp` directory and then it searches in the web module's `WEB-INF/classes` directory.

In a default installation, the WebSphere Application Server `temp` directory is typically *profile_root*/`temp`. Figure 1 shows the processing logic of the JSP engine at runtime.

You should not use the **CANCEL appserver_proc_name** command to stop a server. Every time a server is cleanly stopped, these `temp` directories are removed. However, if the server is frequently not stopped cleanly, which happens if you cancel rather than stop the server, these directories are not removed and the HFS used for the temp directory eventually becomes full. You can also prevent this storage problem from occurring if you precompile your JSP files when you install an application or if you use the JspBatchCompiler function to precompile them before they are invoked.



The batch compiler supports the generation of class files in both the WebSphere Application Server `temp` directory and a web module's `WEB-INF/classes` directory, depending on the type of batch compiler target. In addition, the batch compiler enables the generation of class files into any directory on the filesystem, outside of the target application. Generating class files into a web module's `WEB-INF/classes` directory enables you to deploy the web module as a self-contained web application archive (WAR) file, or a WAR file inside an enterprise archive (EAR) file. The following table shows the batch compiler's behavior when compiling class files.

Table 113. Batch compiler behavior. Batch compiler behavior when compiling class files

	ear.path or war.path supplied	**enterpriseApp.name supplied**
compileToDir not supplied; *compileToWebInf* not supplied, or is true	The class files are compiled into the web module's `WEB-INF/classes` directory.	The class files are compiled into the web module's `WEB-INF/classes` directory.
compileToDir not supplied; *compileToWebInf* is false	The class files are compiled into the web module's `WEB-INF/classes` directory.	The class files are compiled into the WebSphere Application Server temp directory, usually *profile_root*/`temp`.

Table 113. Batch compiler behavior (continued). Batch compiler behavior when compiling class files

	ear.path or war.path supplied	enterpriseApp.name supplied
compileToDir is supplied; *compileToWebInf* not supplied, or is either true or false	The class files are compiled into the directory indicated by *compileToDir*.	The class files are compiled into the directory indicated by *compileToDir*.

Web container configuration for JavaServer Pages static file access

The web container searches for static files and JavaServer Pages (JSP) files in up to four different locations, depending on application configuration. This search is relevant to finding the appropriate resource to serve an inbound request and for results returned from the following APIs:

```
URL ServletContext.getResource(String path)
Set ServletContext.getResourcePaths(String path)
```

Attention: Static files are only searched when the **fileServing** property is enabled and the application does not include a `/*` servlet mapping. Also, JSP files include files with the following extensions, in addition to any other patterns that are defined in the web.xml file

- .jsp
- .jspx
- .jsw
- .jsv

The four locations, in priority order, are as follows:

Application WAR directory

The web container first searches the application WAR directory for a requested resource. However, you cannot serve resources under the WEB-INF or META-INF directories for inbound requests, although they are accessible using the getResource() and getResourcePaths() methods.

Pre-fragment document roots

The web container searches any application defined pre-fragment document roots second. Define a pre-fragment document root in the `ibm-web-ext.xmi` file, located in the WEB-INF directory for the application. Define a pre-fragment document root for static files or JSP files, as follows:

```
Static files:
<fileServingAttributes xmi:id="<user-provided name>"
 name="preFragmentExtendedDocumentRoot"
 value="<user provided value>"/>
```

```
JSP files:
<jspAttributes xmi:id="<user provided name>" name="preFragmentExtendedDocumentRoot"
 value=<user provided value>"/>
```

The *<user provided name>* is a comma-separated list of directories that can include JAR or ZIP files from which static files or JSP files can be accessed. The list can be fully qualified or relative to the application ear directory. The attributes can be specified with identical values. Further, you can specify the same values by different applications.

Attention: Pre-fragment document roots is a new function in WebSphere Application Server Version 8.5.

META-INF/resources directories of fragments under the WEB-INF/Lib directory in the application WAR file

After searching pre-fragment document roots, the web container searches web fragments. A web

fragment comprises a JAR file in an application WEB-INF/lib directory. The JAR might include static resources in a META-INF/resources directory that are defined within the JAR file. To prevent the web container from searching META-INF/resources directories, set the com.ibm.ws.webcontainer.SkipMetaInfResourcesProcessing web container custom property to `true`. The default value for the custom property is false.

```
com.ibm.ws.webcontainer.SkipMetaInfResourcesProcessing = true
```

Attention: META-INF/resources directories of fragments under the WEB-INF/Lib directory in the application WAR file is a new function in WebSphere Application Server Version 8.5.

Extended document roots

Finally, the web container searches any application defined as extended document roots. Extended document roots were available in previous releases and are defined in the `ibm-web-ext.xmi` file, which is located in the application WEB-INF directory. Define extended document root for static files or JSP files, as follows:

```
Static files:
<fileServingAttributes xmi:id="<user provided name>" name="extendedDocumentRoot"
 value="<user provided value>"/>
```

```
JSP files:
<jspAttributes xmi:id="<user provided name>" name="extendedDocumentRoot"
 value=<user provide value>"/>
```

```
com.ibm.ws.webcontainer.SkipMetaInfResourcesProcessing = true
```

The *<user provided name>* is a comma-separated list of directories that can include JAR or ZIP files from which static files or JSP files can be accessed. The list can be fully qualified, or relative to the application EAR directory. You can specify both attributes with identical values. Also, different applications can specify the same values.

Attention: You might set the com.ibm.ws.webcontainer.ServeWelcomeFileFromExtendedDocumentRoot custom property to **true** to enable serving static welcome pages from a static file extended document root. The default value for the custom property is **false**.

```
com.ibm.ws.webcontainer.ServeWelcomeFileFromExtendedDocumentRoot = true
```

When the com.ibm.ws.webcontainer.ServeWelcomeFileFromExtendedDocumentRoot property is set to **true**, the web container searches in a static file extended document root for a static welcome file to serve for a request comprising a valid partial URL. For example, if a request specifies only the application context, and a welcome file is specified as index.html, the property must be set if index.html is to be served from a static file extended document.

```
com.ibm.ws.webcontainer.enablepartialurltoextendeddocumentroot = true
```

The default value is **false**.

If this property is set to **true**, the web container includes the contents of a static file extended document root when determining whether an inbound request is for a valid partial URL. For example, if a request specifies a URL that ends with `/<application context>/<text>` and `<text>` does not map to a servlet or static file, the web container considers this URL a valid partial URL

only if <text> is a valid directory of the application. As a result if directory <text> exists only in a static file extended document root, this property must be set for the URI to be considered a valid partial URI.

Important: These properties are not required for static file pre-fragment document roots. Also, when determining a valid partial URL, the web container does not consider JSP pre-fragment and extended document roots.

Packages and directories for generated .java and .class files

By default, the `.java` files for all JavaServer Pages (JSP) files are generated with the package statement, `package com.ibm._jsp;`. The JSP engine's class loader knows how to load JSP classes when they are all in the same package. The `.java` files are located in the filesystem within a directory structure mirroring the JSP source directory structure.

If the JSP engine configuration parameter **useFullPackageNames** is set to true, the `.java` files are generated with the package statement

```
Package _ibmjsp.<directory structure in which the jsp is located>;
```

The usage of full package names enables the configuration of a JSP as a servlet in the `web.xml` file. Refer to the JSP class loading settings topic for more information. The table later in this section gives examples of packages and directory structures for generated `.java` and `.class` files.

Table 114. Packages and directory structures for generated .java and .class files. Examples of packages and directory structures for generated .java and .class files.

Directory		File name		Location of .java or .class files in file system
JSP file	default	useFullPackageNames=true	default	useFullPackageNames=true
/myJsp.jsp	com.ibm._jsp	_ibmjsp	/	/_ibmjsp
/jspFiles/ jspOne.jsp	com.ibm._jsp	_ibmjsp.jspFiles	/jspFiles	/_ibmjsp/jspFiles
/dir with spaces/jspTwo.jsp	com.ibm._jsp	_ibmjsp.dir_20_with_20_spaces	/dir with spaces	/_ibmjsp/ dir_20_with_20_spaces

Generated .java files:

When the JSP engine's **keepgenerated** configuration parameter is set to true, the `.java` file that is generated for JavaServer Pages (JSP) is retained. The `.java` file contains information that is useful in debugging.

Dependency information

In the `.java` file, immediately following the class declaration, an array of dependent files is defined, if the source JSP has any dependencies. There are three types of files that are tracked as dependencies:

1. Files that are statically included in the JSP
2. Tag files that are used by the JSP, but only tag files that are not in Java Archive (JAR) files
3. TLD files that are used by the JSP, but only TLDs that are not in JAR files

This array is always generated, but the JSP engine uses it, in determining whether a JSP needs to be recompiled, only when the trackDependencies parameter is set to true.

In the example below, three JSP fragments, one TLD and one tag file are dependencies of the JSP jsp1.jsp. There are three parts to each array entry:

1. The path to the dependency, relative to the web module's context root. For example: `/dir1/frag1.jspf`

2. The long value representing the time the file was last modified. For example: 1082407108000
3. The String representation of the long value. For example: Mon Apr 19 16:38:28 EDT 2004

```
public final class _jsp1 extends com.ibm.ws.jsp.runtime.HttpJspBase
  implements com.ibm.ws.jsp.runtime.JspClassInformation {

  private static String[] _jspx_dependants;
  static {
  _jspx_dependants = new String[5];
  _jspx_dependants[0] = "/Banner.jspf^1082407108000^Mon Apr 19 16:38:28 EDT 2004";
  _jspx_dependants[1] = "/Footer.jspf^1077657462000^Tue Feb 24 16:17:42 EST 2004";
  _jspx_dependants[2] = "/dir1/frag1.jspf^1035396680000^Wed Oct 23 14:11:20 EDT 2002";
  _jspx_dependants[3] = "/utility.tld^1080069938000^Tue Mar 23 14:25:38 EST 2004";
  _jspx_dependants[4] = "/WEB-INF/tags/top.tag^1065440490000^Mon Oct 06 07:41:30 EDT 2003";
  }
```

Version, JSP engine options, and WEB.XML information

The generated .java source contains a comment that lists information about the file which is located at the bottom of the generated file. This information includes:

- The date and time the `.java` file was generated
- The version, build number and build date of the WebSphere Application Server on which the `.java` file was generated
- The values of the JSP engine configuration parameters that were in effect when the file was generated
- The values of any `<jsp-config>` elements in the `web.xml` file that pertained to the source JSP file.

```
/*
profile_root/AppSrv01/installedApps/MyCell/sampleApp.ear/examples.war/WEB-INF/classes/_ibmjsp/_jsp1.java
was generated @ Wed May 03 10:05:56 EDT 2006IBM WebSphere Application Server - ND, 6.1.0.0
    Build Number: o0441.04
    Build Date: 05/01/06*******************************************************
The JSP engine configuration parameters were set as follows:

classDebugInfo =           [false]
debugEnabled =             [false]
deprecation =              [false]
compileWithAssert =        [false]
jdkSourceLevel =           [13]disableJspRuntimeCompilation =[false]
extendedDocumentRoot =     [null]
ieClassId =                [clsid:8AD9C840-044E-11D1-B3E9-00805F499D93]
keepGenerated =            [true]



outputDir =                [C:/WebSphere_6.0/AppServer/profiles/AppSrv01/installedApps/MyCell/
                             sampleApp.ear/examples.war/WEB-INF/classes]
reloadEnabled =            [true]
reloadEnabledSet =         [true]
reloadInterval =           [5000]
trackDependencies =        [false]
usePageTagPool =           [false]
useThreadTagPool =         [true]
useImplicitTagLibs =       [true]
verbose =                  [false]
looseLibMap =              [null]
useJikes =                 [false]
useFullPackageNames =      [true]
translationContextClass =  [null]
extensionProcessorClass =  [null]
javaEncoding =             [UTF-8]
autoResponseEncoding =     [false]

******************************************************
The following JSP Configuration Parameters were obtained from web.xml:
```

```
prelude list = [[]]
coda list = [[]]
elIgnored = [false]
pageEncoding = [null]
isXML = [false]
scriptingInvalid = [false]
*/
```

JSP batch compilation

As an IBM enhancement to JavaServer Pages (JSP) support, IBM WebSphere Application Server provides a batch JSP compiler that allows JSP page compilation before application deployment. The batch compiler validates the syntax of JSP pages, translates the JSP pages into Java source files, and compiles the Java source files into Java servlet class files. The batch compiler also validates tag files and generates their Java implementation classes.

Batch compilation of JSP pages in a predeployed application simplifies the deployment process and improves the runtime performance of JSP page by eliminating first-request compilations. The batch compiler also operates on enterprise applications that have been deployed into WebSphere Application Server.

The JSP batch compiler works on web modules that support Servlet 2.2 and later. The batch compiler works on JSP pages written to the JSP 2.1 specification or previous specifications back to JSP 1.0. It recognizes a Servlet 2.5 or later deployment descriptor, `web.xml`, and can use any jsp-config elements that it may contain. In a Servlet 2.3 (JSP 1.2) or Servlet 2.2 (JSP 1.1) deployment descriptor the batch compiler recognizes and uses any taglib elements that the descriptor may contain.

Batch compiling makes the first request for a JSP page much faster because the JSP page is already translated and compiled into a servlet. Batch compiling is also useful as a fast way to resynchronize all of the JSP pages for an application.

The batch compiler supports the generation of class files in both the WebSphere Application Server `temp` directory and a web module's `WEB-INF/classes` directory, depending on the type of batch compiler target. In addition, the batch compiler enables generation of class files into any directory on the filesystem, outside the target application. Generating class files into a web module's `WEB-INF/classes` directory enables the web module to be deployed as a self-contained WAR file, or a WAR inside an EAR.

Also, you can use shared libraries with the JSP batch complier. When you use the JSP batch compiler, you must either add the JAR to the WAR in the `<WEB-INF>/lib` directory, or add the JAR to the JVM class path to use shared libraries.

JSPBatchCompiler command:

The batch compiler validates the syntax of JavaServer Pages, translates the JSP pages into Java source files, and compiles the Java source files into Java Servlet class files. The batch compiler also validates tag files and generates their Java implementation classes. Use this function to batch compile your JSP files and thereby enable faster responses to the initial client requests for the JSP files on your production web server.

The batch compiler can be executed against compressed or expanded enterprise archive (EAR) files and web application archive (WAR) files, as well as enterprise applications and web modules that have been deployed into WebSphere Application Server. When the target is a deployed enterprise application, the server does not need to be running to execute the batch compiler. If the batch compiler is executed while the target sever is running, the server is not aware of an updated class file and does not load that class file unless the enterprise application is restarted. When the target is a compressed EAR file or WAR file, the batch compiler must expand it before executing.

Processing of web modules

The batch compiler operates on one web module at a time. If the target is either an EAR file or an installed enterprise application that contains more than one web module, the batch compiler operates on each web module individually. This is done because JSP pages are configured on a web module basis, through the web module's web.xml deployment descriptor file. Within a web module, the batch compiler processes one directory at a time. It validates and translates each JSP page individually, and then invokes the Java compiler for the entire group of generated Java sources files in that directory. If one JSP page fails during the Java compilation phase, the Java compiler might not create class files for most or all of the JSP pages that successfully compiled in that directory.

JSP file extensions

The batch compiler uses four things to determine what file extensions it should process:

1. Standard JSP file extensions
   - *.jsp
   - *.jspx
   - *.jsw
   - *.jsv
2. The url-pattern property of the jsp-property-group elements in the deployment descriptor file in Servlet 2.4 web modules
3. The `jsp.file.extensions` JSP engine configuration parameter (for pre-Servlet 2.4 web modules)
4. The batch compiler configuration parameter `jsp.file.extensions`

The standard extensions are always used by the batch compiler. If the web module contains a Servlet 2.4 deployment descriptor, the batch compiler also processes any url-patterns found within the jsp-config element. If the batch compiler target contains the JSP engine configuration parameter `jsp.file.extensions`, then those extensions are also processed. If the batch compiler configuration parameter `jsp.file.extensions` is present, the extensions given are also processed and will override the JSP engine configuration parameter `jsp.file.extensions`.

It is a good idea to give JSP 'fragments' an extension that is not processed by the batch compiler. Statically-included fragments that do not stand alone generate translation or compilation errors if processed. The JSP 2.0 Specification suggests that you use the extension `.jspf` for such files.

Batch compiler command

Both a Windows batch file, `JspBatchCompiler.bat` and UNIX shell script `JspBatchCompiler.sh` for running the batch compiler from the command line are found in the `{WAS_ROOT}/bin` directory. An Ant task is also available for executing the batch compiler using Ant. See the topic, Batch Compiler Ant Task for additional information.

The batch compiler target is the only required parameter. The target is one of -ear.path, -war.path or -enterpriseapp.name.

```
JspBatchCompiler -ear.path | -war.path | -enterpriseapp.name <name>
   [-response.file <filename>]
   [-webmodule.name <name>]
   [-filename <jsp name | directory name>
   [-recurse <true | false>]
   [-config.root <path>]
   [-cell.name <name>]
   [-cluster.name <name>]   [-node.name <name>]
   [-server.name <name>]
   [-profileName <name>]
   [-extractToDir <path>]
   [-compileToDir <path>]
```

```
[-compileToWebInf <true | false>]
   [-compileToWebInf <true | false>]
   [-compileAfterFailure <true | false>]
[-translate <true | false>]
[-compile <true | false>]
[-removeTempDir <true | false>]
[-forceCompilation <true | false>]
[-useFullPackageNames <true | false>]
[-trackDependencies <true | false>]
[-createDebugClassfiles <true | false>]
[-keepgenerated <true | false>]
[-keepGeneratedclassfiles <true | false>]
[-usePageTagPool <true | false>]
[-useThreadTagPool <true | false>]
[-classloader.parentFirst <true | false>]
[-classloader.singleWarClassloader <true | false>]
[-additional.classpath <classpath to additional JAR files and classes>]

[-verbose <true | false>]
[-deprecation <true | false>]
[-javaEncoding <encoding>
[-jdkSourceLevel <13 | 14 | 15 | 16 | 17>]
[-compilerOptions <space-separated list of java compiler options>]
[-useJikes <true | false>]
[-jsp.file.extensions <file extensions to process>]
[-log.level <SEVERE | WARNING | INFO | CONFIG | FINE | FINER | FINEST | OFF>]
*** See batchcompiler.properties.default in WAS_ROOT/bin for more information. ***
*** See JspCBuild.xml in WAS_ROOT/bin for information about the public WebSphere Ant task JspC. ***
```

The batch compiler is aware of three groups of configuration parameters:

1. JSP engine configuration parameters for a web module.

   Refer to the JSP engine configuration parameters topic.
2. Batch compiler response file configuration parameters.

   These are the parameters that are found in a batch compiler response file. See -response.file, below.
3. Batch compiler command line configuration parameters.

   These are the parameters entered on the command line when running the batch compiler.

The batch compiler looks at all three groups of configuration parameters in order to determine which value for a parameter is used when compiling JSP pages. When resolving the value for a given parameter, the precedence is:

1. If the parameter is found on the command line, its value is used.
2. If the parameter is not found on the command line, the batch compiler looks for the parameter in a response file named on the command line.
3. If no response file is named, or if the parameter is not found therein, the batch compiler looks for the parameter in the web module's JSP engine configuration parameters.

If a configuration parameter is not found among these three groups, then a default value is used. The default values for the configuration parameters are given below along with the description of the parameters.

With one exception, these parameters are not case sensitive; -profileName is case sensitive. If the values specified for these arguments are comprised of two or more words separated by spaces, you must add quotation marks around the values.

The batch compiler does not create, or set the values of, equivalent JSP engine parameters. This means that if a JSP page in a deployed Web module is modified and is recompiled by the JSP engine at run time, the JSP engine's configuration parameters will determine the engine's behavior. For example, if you use the batch compiler to compile a web module and you use the -useFullPackageNames true option, the JSP

files will be compiled to support that option. But the JSP engine parameter useFullPackageNames must also be set to true in order for the JSP runtime to be able to load the compiled JSP pages. If JSP pages are modified in a deployed web module, then the engine's parameters should be set to the same values used in batch compilation.

To use the JSP batch compiler, enter one of the following commands on a single line at an operating system command prompt.:

- *ear.path* | *war.path* | *enterpriseapp.name*

  Represents the full path to a single compressed or expanded enterprise application archive (EAR) file or web application archive (WAR) file, or the name of the deployed enterprise application that you want to compile. For example:
  - JspBatchCompiler -ear.path /myhfs/myprojects/sampleApp.ear
  - JspBatchCompiler -war.path c:\myWars\examples.war
  - JspBatchCompiler -enterpriseapp.name myEnterpriseApp -webmodule.name my.war -filename aDir/main.jsp
- `response.file`

  Specifies the path to a file that contains configuration parameters used by the batch compiler. The `response.file` is used only if it is given on the command line; it is ignored if it is present in a response file.

  In a default installation, the template response file, batchcompiler.properties.default, is found in the `{WAS_ROOT}/bin` directory. Copy this template to create your own response files containing defaults for the parameters in which you are interested. All the required and optional parameters (except response.file) can be configured in a response file. For example: JspBatchCompiler -response.file c:\myproject\batchc.props

  Default : null
- *webmodule.name*

  Represents the name of the specific web module that you want to batch compile. If this argument is not set, all web modules in the enterprise application are compiled. This parameter is used only when *ear.path* or *enterpriseapp.name* is given. This parameter is useful when JSP pages in a specific web module within a deployed enterprise application need to be regenerated, because all shared library dependencies are picked up.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -webmodule.name myWebModule.war

  Default: All web modules in an EAR file or enterprise application are compiled if this parameter is not given.
- *filename*

  Represents the name of a single JSP file that you want to compile. If this argument is not set, all files in the web module are compiled. Alternatively, if *filename* is set to the name of a directory, only the JSP files in that directory and that directory's child directories are complied. The name is relative to the context root of the web module.

  Example 1: If you want to compile the file, *myTest.jsp*, and it is found in */subdir/myJSPs*, you would enter *-filename /subdir/myJSPs/myTest.jsp*.

  Example 2: If you want to compile all JSP files in */subdir/myJSPs* and its child directories, you would enter *-filename subdir/myJSPs*.

  Default: All JSP files in the web module are compiled. Entering *-filename /* is equivalent to the default.
- recurse

  Determines whether subdirectories beneath the target directory are processed. This parameter is used only when the *filename* parameter is given. Set value to `false` to process only the directory named *filename* parameter; and not its subdirectories.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -filename /subdir1 -recurse false.

  Default: true; All directories beneath the target directory are processed.
- *config.root*

Specifies the location of the WebSpehere Application Server configuration directory. This parameter is used only when *enterpriseapp.name* is given.

Default: *{WAS_ROOT}/profiles/profilename/config*

- *cell.name*

  Specifies the name of the cell in which the application is deployed. This parameter is used only when *enterpriseapp.name* is given.

  Default: The default is obtained from the profile script that is used. The symbolic name of this variable is WAS_CELL.

- *cluster.name*

  Specifies the name of the cluster in which the application is deployed. This parameter provides the batch compiler with access to cluster scoped shared libraries, and is used only when *enterpriseapp.name* is given.

  Default: The default is obtained from the profile script that is used. The symbolic name of this variable is WAS_CLUSTER.

- *node.name*

  Specifies the name of the node in which the application is deployed. This parameter is used only when *enterpriseapp.name* is given.

  Default: The default is obtained from the profile script that is used. The symbolic name of this variable is WAS_NODE.

- *server.name*

  Represents the name of the server in which the application is deployed. This parameter is used only when *enterpriseapp.name* is given.

  Default: server1

- *profileName*

  Specifies the name of the profile you want to use. This parameter is used only when the *enterpriseapp.name* or *-ear*. path is given.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -profileName AppServer-3

  Default: The default profile is used. This is obtained from the file `setupCmdLine` script in the `install_root/bin` directory. The symbolic name is DEFAULT_PROFILE_SCRIPT.

- *extractToDir*

  Specifies the directory into which predeployed enterprise archive (EAR) files and web application archive (WAR) files will be extracted before the batch compiler operates on them. This parameter is ignored when *enterpriseapp.name* is given. The extractToDir parameter is used as described in the table below.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -extractToDir c:\myTempDir.

  Use-case: You must extract a compressed archive before it is batch compiled. You can also extract an expanded archive to a new directory as well. In both cases, extraction leaves the original archive untouched, which may be useful while development is underway.

Table 115. extractToDir. Default values

	Expanded archive	**Compressed archive**
extractToDir supplied	The batch compiler extracts the archive to extractToDir before operating on it. If a file or directory of the same name as the archive already exists in the extractToDir, the batch compiler removes the archive completely before extracting that archive. If the batch compiler exits with no errors, it compresses the archive in place in the extractToDir, even if the original EAR file or WAR file was expanded. If errors are encountered during compilation, the EAR file or WAR file is left in the expanded state even if the original EAR file or WAR file was compressed.	

Table 115. extractToDir (continued). Default values

	Expanded archive	**Compressed archive**
extractToDir not supplied	The batch compiler operates on the EAR file or WAR file in place (does not extract it to another directory) and the archive remains expanded after the batch compiler finishes.	The batch compiler extracts the archive to the directory returned by the JVM property “java.io.tmpdir”. The rest of the behavior described above, when extractToDir is supplied, is the same in this case.

The default is *server1*.

- *compileToDir*

  Specifies the directory into which JSP pages are translated into Java source files and compiled into class files. This directory can be anywhere on the filesystem, but the batch compiler's default behavior is usually adequate. The batch compiler's behavior when compiling class files is described in the table below

  Example:: JspBatchCompiler -enterpriseApp.name sampleApp -compileToDir c:\myTargetDir

  Use-case: This parameter enables you to generate the Java and class files into a directory outside of the target, which may be useful if you want to compare the newly generated files with their previous versions which remain untouched within the target.

Table 116. compileToDir. Default values

	ear.path or war.path supplied	**enterpriseApp.name supplied**
compileToDir not supplied; compileToWebInf not supplied, or is true	The class files are compiled into the web module's `WEB-INF/classes` directory	The class files are compiled into the web module's `WEB-INF/classes` directory.
compileToDir not supplied; compileToWebInf is false	The class files are compiled into the web module's `WEB-INF/classes` directory.	The class files are compiled into the WebSphere Application Server temp directory (usually *{WAS_ROOT}/temp*).
compileToDir is supplied; compileToWebInf not supplied, or is either true or false	The class files are compiled into the directory indicated by compileToDir.	The class files are compiled into the directory indicated by compileToDir.

- *compileToWebInf*

  Specifies whether the target directory for the compiled JSP class files should be the web module's `WEB-INF/classes` directory. This parameter is used only when *enterpriseApp.name* is given, and it is overridden by *compileToDir* if *compileToDir* is given.

  The batch compiler's default behavior is to compile to the web module's `WEB-INF/classes` directory. The batch compiler's behavior when compiling class files is described in the table above.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -compileToWebInf false.

  Use-case: Set this parameter to false when enterpriseApp.name is supplied and you want the class files to be compiled to the WebSphere Application Server temp directory instead of the web module's `WEB-INF/classes` directory. Recommendation: if this parameter is set to false, set forceCompilation to true if there are any JSP class files in the `WEB-INF/classes` directory.

  Default: true; see the table above.

- *compileAfterFailure*

  Specifies whether the JDK JSP batch compiler continues to compile the other JavaServer Pages (JSP) files in the current directory if one or more of the JavaServer Pages (JSP) files in that directory cannot be complied. Typically when one of the files cannot be compiled, the JSP batch complier skips all of the remaining JSPs in that directory, and starts to compile the files in the next directory.

  If you set this parameter to true, you must also specify the useJDKCompiler parameter and set that parameter to true.

Example: JspBatchCompiler -enterpriseApp.name sampleApp -useJDKCompiler true –compileAfterFailure false.

Use-case: Set this parameter to true if you want the JSP batch compiler compile the other JavaServer Pages (JSP) files in the current directory even if one or more of the JSP files in that directory cannot be complied.

Default: false

- *forceCompilation*

  Specifies whether the batch compiler is forced to recompile all JSP resources regardless or whether the JSP page is outdated.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -forceCompilation true.

  Use-case: Especially useful when creating an archive for deployment, to make sure all JSP classes are up to date.

  Default: false

- *useFullPackageNames*

  Specifies whether the batch compiler generates full package names for JSP classes. The default is to generate all JSP classes in the same package. The JSP engine's class loader knows how to load JSP classes when they are all in the same package. The default has the benefit of generating smaller file-system paths. Full package names have the benefit of enabling the configuration of precompiled JSP class files as servlets in the `web.xml` file without use of the `jsp-file` attribute, resulting in a single class loader (the web application's class loader) being used to load all such JSP classes. Similarly, when the JSP engine's configuration attributes `useFullPackageNames` and `disableJspRuntimeCompilation` are both true, a single class loader is used to load all JSP classes, even if the JSP pages are not configured as servlets in the `web.xml` file.

  When `useFullPackageNames` is set to true, the batch compiler generates a file called `generated_web.xml` in the web module's `WEB-INF` directory. This file contains servlet configuration information for each JSP page that is successfully translated and compiled. The information can optionally be copied into the web module's `web.xml` file so that the JSP pages are loaded as servlets by the web container. Note that if a JSP page is configured as a servlet in this way, no reloading of the JSP page is done at run time if the JSP page is modified. This is because the JSP page is treated as a regular servlet and requests for it do not pass through the JSP engine.

  Example: JspBatchCompiler –enterpriseApp.name sampleApp –useFullPackageNames true

  Use-case: Enables JSP classes to be loaded by a single class loader.

  Default: false

- *removeTempDir*

  Specifies whether the web module's temp directory is removed. The batch compiler by default generates JSP class files into a web module's `WEB-INF/classes` directory. JSP class files are generated into the `temp` directory at run time if a JSP page is modified and JSP reloading is enabled. By batch compiling all the JSP pages in a web module and also removing the `temp` directory, disk resources are preserved. You can only use the removeTempDir parameter when -enterpriseApp.name is given.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -removeTempDir true.

  Use-case: Free up disk space by clearing out a web application's `temp` directory.

  Default: false

- *translate*

  Specifies whether JSP pages are translated and compiled. Set translate to false if you do not want JSP pages to be translated and compiled. You must use this option in conjunction with `-removeTempDir` to tell the batch compiler to remove only the `temp` directory and to do no further processing.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -translate false -removeTempDir true.

  Use-case: Free up disk space by clearing out a web application's `temp` directory, without invoking JSP processing.

  Default: true

- *compile*

Specifies whether JSP pages go through the Java compilation phase. Set compile to `false` if you do not want JSP pages to go through the Java compilation phase.

Example: JspBatchCompiler -enterpriseApp.name sampleApp -compile false

Use-case: If you only want JSP pages to be syntax-checked, set -compile to false. You can set -keepgenerated to true if you want to see the `.java` files that are generated during the translation phase.

Default: true

- *trackDependencies*

  Specifies whether the batch compiler recompiles a JSP page when any of its dependencies have changed, even if the JSP page itself has not changed. Tracking dependencies incurs a significant runtime performance penalty because the JSP Engine checks the filesystem on every request to a JSP page to see if any of its dependencies have changed. The dependencies tracked by WebSphere Application Server are :
  1. Files statically included in the JSP page
  2. Tag files used by the JSP page (excluding tag files that are in JAR files)
  3. TLD files used by the JSP page (excluding TLD files that are in JAR files)

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -trackDependencies true.

  Use-case: Useful in a development environment.

  Default: false
- *createDebugClassfiles*

  Specifies whether the batch compiler generates class files that contain SMAP information, as per JSR 45, Debugging support for Other Languages.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -createDebugClassfiles true

  Use-case: Use this parameter when you want to be able to debug JSP pages in your JSR 45-compliant IDE.

  Default: false
- *keepgenerated*

  Specifies whether the batch compiler saves or erases the generated Java source files created during the translation phase.

  If set to `true`, WebSphere Application Server saves the generated `.java` files used for compilation on your server. By default, this argument is set to `false` and the `.java` files are erased after the class files have compiled.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -keepgenerated true

  Use-case: Use this parameter when you want to review the Java code generated by the batch compiler.

  Default: false
- *keepGeneratedclassfiles*

  Specifies whether the batch compiler saves or erases the class files generated during the compilation of Java source files.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -keepGeneratedclassfiles false -keepgenerated false

  Use-case: Set this parameter to false if you only want to see if there are any translation or compilation errors in your JSP pages. If paired with -keepgenerated false, this parameter results in all generated files being removed before the batch compiler completes.

  Default: true
- *usePageTagPool*

  Enables or disables the reuse of custom tag handlers on an individual JSP page basis.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -usePageTagPool true

  Use-case: Use this parameter to enable JSP-page-based reuse of tag handlers.

  Default: false
- *useThreadTagPool*

Enables or disables the reuse of custom tag handlers on a per request thread basis per web module.

Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -useThreadTagPool true

Use-case: Use this parameter to enable web module-based reuse of tag handlers.

Default: false

- *classloader.parentFirst*

  Specifies the search order for loading classes by instructing the batch compiler to search the parent class loader prior to application class loader. This parameter is only used when *ear.path* or *enterpriseApp.name* is given.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -classloader.parentFirst false

  Use-case: Set this parameter to false when your web module contains a JAR file that is also found in the server lib directory, and you want your web module's JAR file to be picked up first.

  Default: true

- *classloader.singleWarClassloader*

  Specifies whether to use one class loader per enterprise archive (EAR) file or one class loader per web application archive (WAR) file. Used only when *ear.path* or *enterpriseApp.name* is given.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -classloader.singleWarClassloader true

  Use-case: Set this parameter to true when a Web module depends on JAR files and classes in another web module in the same enterprise application.

  Default: false; One class loader is created per WAR file with no visibility of classes in other web modules.

- *additional.classpath*

  Specifies additional class path entries to be used when parsing and compiling JSP pages. This parameter is used only when *war.path* is given. When *war.path* is the target, WebSphere Shared Libraries are not picked up by the batch compiler. Therefore, if your WAR file relies on, for example, a JAR file that is configured in WebSphere Application Server as a shared library, then use this option to point to that JAR file. In addition, if you give *war.path* and also use the *-extractToDir* parameter, then any JAR files that are in the WAR file's manifest class-path is not added to the class path (since the WAR file has now been extracted by itself outside the EAR file in which it resides). Use *-additional.classpath* in this case to point to the necessary JAR files. Add the full path to needed resources, separated by your system-dependent path separator.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -additional.classpath c:\myJars\someJar.jar;c:\myClasses

  Use-case: Use this parameter to add to the class path JAR files and classes outside of your WAR file. At run time, these same JAR files and classes have to be made available through the standard WebSphere Application Server configuration mechanisms.

  Default: null

- *verbose*

  Specifies whether the batch compiler should generate verbose output while compiling the generated sources.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -verbose true

  Use-case: Set this parameter to true when you want to see Java compiler class loading and other messages.

  Default: false

- *deprecation*

  Indicates the compiler should generate deprecation warnings while compiling the generated sources.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -deprecation true

  Use-case: Set this parameter to true when you want to see Java compiler deprecation messages.

  Default: false

- *javaEncoding*

  Specifies the encoding that will be used when the .java file is generated, and when it is compiled by the Java compiler. When -javaEncoding is set, that encoding is passed to the java compiler via the -encoding argument. Note that encoding is not supported by Jikes.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -javaEncoding Shift-JIS

  Use-case: Set this parameter when the page encoding of your JSP pages is not UTF-8 compatible.

  Default value: UTF-8.

- *jdkSourceLevel*

  This JSP engine parameter was introduced in WebSphere Application Server version 6.1 to support JDK 5. Use this parameter instead of the compileWithAssert parameter, although compile WithAssert still works in version 6.1.

  The default value for this parameter is 16. This parameter requires regeneration of Java source. The following are jdkSourceLevel paramater values:

  – 13 - This value will disable all new language features of JDK 1.4, JDK 5.0, JDK 6.0, and JDK 7.0.
  – 14 - This value will enable the use of the assertion facility and will disable all new language features of JDK 5.0, JDK 6.0, and JDK 7.0.
  – 15 - This value will enable the use of the assertion facility and will disable all new language features of JDK 6.0 and JDK 7.0.
  – 16 - This value will enable the use of the assertion facility and will disable all new language features of JDK 7.0.
  – 17 - This value will enable the use of the new features of JDK 7.0.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -jdkSourceLevel 14

  Use-case: Set this parameter when you want to enable or disable the language features of JDK 1.4 , JDK 5.0, JDK 6.0, and JDK 7.0

  Default value: 16

- *compilerOptions*

  Specifies a list of strings to be passed on the Java compiler command. This is a space-separated list of the form “arg1 arg2 argn”.

  Example: JspBatchCompiler -war.path c:\myproject\examples.war -compilerOptions “-bootclasspath <path>”

  Use-case: Use this parameter if you need Java compiler arguments other than verbose, deprecation and Assert facility support.

  Default: null

- *useJikes*

  Specifies whether Jikes should be used for compiling Java sources. NOTE: Jikes is not shipped with WebSphere Application Server.

  Example: JspBatchCompiler -ear.path c:\myproject\sampleApp.ear -useJikes true

  Use-case: Set this parameter to true in order for the batch compiler to use Jikes as the Java compiler.

  Default value: false

- *jsp.file.extensions*

  Specifies the file extensions to be processed by the batch compiler. This is a semicolon- or colon-separated list of the form “*.ext1;*.ext2:*.extn”. Note that this parameter is not necessary for Servlet 2.4 web applications because the url-pattern property of the jsp-property-group elements in the deployment descriptor can be used to identify extensions that should be treated as JSP pages.

  Example: JspBatchCompiler -enterpriseApp.name sampleApp -jsp.file.extensions *jspz;*.jspt

  Use-case: Use this parameter to add additional extensions to the be processed by the batch compiler.

  Default: null. See section, “JSP file extensions”, in this topic for additional information.

- *log.level*

Specifies the level of logging that is directed to the console during batch compilation. Values are SEVERE | WARNING | INFO | CONFIG | FINE | FINER | FINEST | OFF

Example: JspBatchCompiler -enterpriseApp.name sampleApp -log.level FINEST

Use-case: Set this parameter higher or lower to control logging output. FINEST generates the most output useful for debugging.

Default: CONFIG

Batch compiler ant task:

The ant task **JspC** exposes all the batch compiler configuration options. It runs the batch compiler under the covers. It is backward compatible with the WebSphere Application Server 5.x version of the **JspC** ant task. The following table lists all of the ant task attributes and their batch compiler equivalents.

Table 117. Ant task attributes and their batch compiler equivalents. JspC attributes and the equivalent batch compiler parameters.

JspC attribute	Equivalent batch compiler parameter
earPath	-ear.path
warPath	-war.path
src Same as warPath, for backward compatiblity	-war.path
enterpriseAppName	-enterpriseapp.name
responseFile	-response.file
webmoduleName	-webmodule.name
fileName	-filename -config.root
configRoot	-config.root
cellName	-cell.name
nodeName	-node.name
serverName	-server.name
profileName	-profileName
extractToDir	-extractToDir
compileToDir same as compileToDir, for backward compatibility	-compileToDir -compileToDir
compileToWebInf	-compileToWebInf
compilerOptions	-compilerOptions
recurse	-recurse
removeTempDir	-removeTempDir
translate	-translate
compile	-compile
forceCompilation	-forceCompilation
useFullPackageNames	-useFullPackageNames
trackDependencies	-trackDependencies
createDebugClassfiles	-createDebugClassfiles
keepgenerated	-keepgenerated
keepGeneratedclassfiles	-keepGeneratedclassfiles
usePageTagPool	-usePageTagPool
useThreadTagPool	-useThreadTagPool
classloaderParentFirst	-classloader.parentFirst
classloaderSingleWarClassloader	-classloader.singleWarClassloader
additionalClasspath	-additional.classpath
classpath same as additionalClasspath, for backward compatibility	-additional.classpath
verbose	-verbose

Table 117. Ant task attributes and their batch compiler equivalents (continued). JspC attributes and the equivalent batch compiler parameters.

JspC attribute	Equivalent batch compiler parameter
deprecation	-deprecation
javaEncoding	-javaEncoding
compileWithAssert	-compileWithAssert
useJikes	-useJikes
jspFileExtensions	-jsp.file.extensions
logLevel	-log.level
wasHome	none
Classpathref	none
jdkSourceLevel	-jdkSourceLevel

Below is an example of a build script with multiple targets, each with different attributes. The following commands are used to launch the script:

On Windows:

```
ws_ant -Dwas.home=%WAS_HOME% -Dear.path=%EAR_PATH% -Dextract.dir=%EXTRACT_DIR%
ws_ant jspc2 -Dwas.home=%WAS_HOME% -Dapp.name=%APP_NAME% -Dwebmodule.name=%MOD_NAME%
ws_ant jspc3 -Dwas.home=%WAS_HOME% -Dapp.name=%APP_NAME% -Dwebmodule.name=%MOD_NAME% -Ddir.name=%DIR_NAME%
```

On UNIX or i5/OS:

```
ws_ant -Dwas.home=$WAS_HOME -Dear.path=$EAR_PATH -Dextract.dir=$EXTRACT_DIR
ws_ant jspc2 -Dwas.home=$WAS_HOME -Dapp.name=$APP_NAME -Dwebmodule.name=$MOD_NAME
ws_ant jspc3 -Dwas.home=$WAS_HOME -Dapp.name=$APP_NAME -Dwebmodule.name=$MOD_NAME -Ddir.name=$DIR_NAME
```

Example build.xml file using the JspC task

```
<project name="JSP Precompile" default="jspc1" basedir=".">
 <taskdef name="wsjspc" classname="com.ibm.websphere.ant.tasks.JspC"/>
 <target name="jspc1" description="example using  a path to an EAR, and extracting the EAR to a directory">
  <wsjspc wasHome="${was.home}"
            earpath="${ear.path}"
            forcecompilation="true"
            extractToDir="${extract.dir}"
            useThreadTagPool="true"
            keepgenerated="true"

  />
 </target>
 <target name="jspc2" description="example using an enterprise app and webmodule">
  <wsjspc wasHome="${was.home}"
            enterpriseAppName="${app.name}"
            webmoduleName="${webmodule.name}"
            removeTempDir="true"
            forcecompilation="true"
            keepgenerated="true"

  />
 </target>
 <target name="jspc3" description="example using an enterprise app, webmodule and specific directory">
  <wsjspc wasHome="${was.home}"
            enterpriseAppName="${app.name}"
            webmoduleName="${webmodule.name}"
            fileName="${dir.name}"
            recurse="false"
            forcecompilation="true"
            keepgenerated="true"

  />
 </target>
</project>
```

Pre-touch tool for compiling and loading JSP files:

When enabled, the pre-touch mechanism causes all JavaServer Pages (JSP) files to be compiled within the web module for which they are configured. You can also configure some or all JSP files to be class loaded and JIT-compiled.

To enable the pre-touch mechanism, use Rational Application Developer to specify the following JSP attributes, which are Assembly Property Extensions for your web module:

- **prepareJSPs (Required)**

  When this attribute is present, all JSP files are compiled at application server startup. This activity runs in a separate thread, allowing the application server to finish other startup actions in parallel. The numeric attribute value represents the minimum size in kilobytes that a JSP file must be in order to also be class loaded and JIT-compiled. The default is 0, which causes all JSP files to be class loaded and JIT-compiled.

  Note: JSP file compilation is different from JIT compilation. JSP compilation generates bytecodes, whereas JIT translates the bytecodes into machine code at run time.

- **prepareJSPAttribute (Optional)**

  The pre-touch mechanism compiles and JIT-compiles JSP files by directly invoking the JSP service method, thus making the JSP file susceptible to incurring exceptions because it is called out of context. Such exceptions are avoided by immediately checking the value of this attribute, causing a quick exit from the service method when the JSP was prepared by this tool. This attribute value is added as a request parameter and is composed of alphanumeric characters that your JSP files do not expect to use during normal initiation.

- **prepareJSPThreadCount (Optional)**

  Set this numeric attribute to the number of threads that you would like this mechanism to start up to compile your JSP files. Since a thread makes use of just one processor, multi-processor systems may better utilize this pre-touch mechanism by specifying a value greater than 1. The default setting for this attribute is 1, representing the number of threads that are created to perform pre-touch processing for this web module.

- **prepareJSPClassload (Optional)**

  Set this attribute to either a whole number or the word *changed*. By entering *changed*, only those JSP filess that have been updated or not previously touched, for example, those JSP files that need to be converted from a `.jsp` file to a `.java` file, are class loaded. By entering a numerical value, for example, 1000, the pre-touch tool starts class loading at the 1000th JSP that it processes and all subsequent JSP files. This is convenient in the event that the application server is stopped when starting the pre-touch tool. You can then check the server logs to see how many JSP files have been processed and update the prepareJSPClassload value accordingly to avoid duplicating work. If a JSP file is not class loaded, it cannot be JIT compiled. As a result, if a JSP file does not satisfy the requirements of the prepareJSPClassload attribute, but satisfies the requirements of the prepareJSPs attribute, the JSP file is compiled if it has been updated, but is not class loaded or JIT compiled.

Batch compiler class path:

The batch compiler builds its class path as shown in the table later in this topic. When the batch compiler target is a web application archive (WAR) file and `war.path` is supplied, the configuration `additional.classpath` parameter is used to give extra class path information.

Table 118. Batch compiler. Batch compiler target.

Location added to class path	enterpriseapp.name	ear.path	war.path
WebSphere Application Server JAR files and classes	yes	yes	yes

Table 118. Batch compiler (continued). Batch compiler target.

Location added to class path	enterpriseapp.name	ear.path	war.path
JAR files listed in manifest class path for a web module	yes	yes	yes, when the target WAR is inside an EAR and –extractToDir is not used; otherwise, no.
Shared libraries	yes	no	no
Web module JAR files and classes	yes	yes	yes
additional.classpath parameter to batch compiler	no	no	yes

Global tag libraries (deprecated)

JavaServer Pages (JSP) tag libraries contain classes for common tasks such as processing forms and accessing databases from JSP files.

Tag libraries encapsulate, as simple tags, core functionality common to many web applications. The Java Standard Tag Library (JSTL) supports common programming tasks such as iteration and conditional processing, and provides tags for:

- manipulating XML documents
- supporting internationalization
- using Structured Query Language (SQL)

Tag libraries also introduce the concept of an expression language to simplify page development, and include a version of the JSP expression language.

A tag library has two parts - a Tag Library Descriptor (TLD) file and a Java archive (JAR) file.

tsx:dbconnect tag JavaServer Pages syntax (deprecated):

Use the <tsx:dbconnect> tag to specify information needed to make a connection to a database through Java DataBase Connectivity (JDBC) or Open Database Connectivity (ODBC) technology.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The <tsx:dbconnect> syntax does not establish the connection. Use the <tsx:dbquery> and <tsx:dbmodify> syntax instead to reference a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file to establish the connection.

When the JSP file compiles into a servlet, the Java processor adds the Java coding for the <tsx:dbconnect> syntax to the servlet service() method, which means a new database connection is created for each request for the JSP file.

This section describes the syntax of the <tsx:dbconnect> tag.

```
<tsx:dbconnect id="connection_id"
    userid="db_user" passwd="user_password"
    url="jdbc:subprotocol:database"
    driver="database_driver_name"
    jndiname="JNDI_context/logical_name">
</tsx:dbconnect>
```

where:

- **id**

Represents a required identifier. The scope is the JSP file. This identifier is referenced by the connection attribute of a <tsx:dbquery> tag.

- **userid**

  Represents an optional attribute that specifies a valid user ID for the database that you want to access. Specify this attribute to add the attribute and its value to the request object.

  Although the userid attribute is optional, you must provide the user ID. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this information in the JSP file.

- **passwd**

  Represents an optional attribute that specifies the user password for the userid attribute. (This attribute is not optional if the userid attribute is specified.) If you specify this attribute, the attribute and its value are added to the request object.

  Although the passwd attribute is optional, you must provide the password. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this attribute in the JSP file.

- **url** and **driver**

  Respresents a required attribute if you want to establish a database connection. You must provide the URL and driver.

  The application server supports connection to JDBC databases and ODBC databases.

  – For a JDBC database, the URL consists of the following colon-separated elements: jdbc, the subprotocol name, and the name of the database to access. An example for a connection to the Sample database included with IBM DB2 is:

    ```
    url="jdbc:db2:sample"
    driver="com.ibm.db2.jdbc.app.DB2Driver"
    ```

  – For an ODBC database, use the Sun JDBC-to-ODBC bridge driver included in their Java2 Software Developers Kit (SDK) or another vendor's ODBC driver.

    The url attribute specifies the location of the database. The driver attribute specifies the name of the driver to use in establishing the database connection.

    If the database is an ODBC database, you can use an ODBC driver or the Sun JDBC-to-ODBC bridge. If you want to use an ODBC driver, refer to the driver documentation for instructions on specifying the database location with the url attribute and the driver name.

    If you use the bridge, the url syntax is `jdbc:odbc:`*`database`*. An example follows:

    ```
    url="jdbc:odbc:autos"
    driver="sun.jdbc.odbc.JdbcOdbcDriver"
    ```

    Note: To enable the application server to access the ODBC database, use the ODBC Data Source Administrator to add the ODBC data source to the System DSN configuration. To access the ODBC Administrator, click the ODBC icon on the Windows NT Control Panel.

- **jndiname**

  Represents an optional attribute that identifies a valid context in the application server Java Naming and Directory Interface (JNDI) naming context and the logical name of the data source in that context. The web administrator configures the context using an administrative client such as the WebSphere Administrative Console.

  If you specify the jndiname attribute, the JSP processor ignores the driver and url attributes on the <tsx:dbconnect> tag.

An empty element (such as <url></url>) is valid.

dbquery tag JavaServer Pages syntax (deprecated):

Use the <tsx:dbquery> tag to establish a connection to a database, submit database queries, and return the results set.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The <tsx:dbquery> tag does the following:
1. References a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file and uses the information the tag provides to determine the database URL and driver. You can also obtain the user ID and password from the <tsx:dbconnect> tag if those values are provided in the <tsx:dbconnect> tag.
2. Establishes a new connection
3. Retrieves and caches data in the results object.
4. Closes the connection and releases the connection resource.

This section describes the syntax of the <tsx:dbquery> tag.

```
<%-- SELECT commands and (optional) JSP syntax can be placed within the tsx:dbquery. --%>
<%-- Any other syntax, including HTML comments, are not valid. --%>
<tsx:dbquery id="query_id" connection="connection_id" limit="value" >
</tsx:dbquery>
```

where:
- **id**

  Represents the identifier of this query. The scope is the JSP file. Use `id` to reference the query. For example, from the <tsx:getProperty> tag, use `id` to display the query results.

  The `id` is a tsx reference to the bean and can be used to retrieve the bean from the page contect. For example, if id is named mySingleDBBean, instead of using:
  – if (mySingleDBBean.getValue("UISEAM",0).startsWith("N"))

  use:
  – com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults bean = (com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults)pageContext. findAttribute("mySingleDBBean"); if (bean.getValue("UISEAM",0).startsWith("N")). . .

  The bean properties are dynamic and the property names are the names of the columns in the results set. If you want different column names, use the SQL keyword for specifying an alias on the SELECT command. In the following example, the database table contains columns named FNAME and LNAME, but the SELECT statement uses the `AS` keyword to map those column names to FirstName and LastName in the results set:

  ```
  Select FNAME, LNAME AS FirstName, LastName from Employee where FNAME='Jim'
  ```
- **connection**

  Represents the identifier of a <tsx:dbconnect> tag in this JSP file. The <tsx:dbconnect> tag provides the database URL, driver name, and optionally, the user ID and password for the connection.
- **limit**

  Represents an optional attribute that constrains the maximum number of records returned by a query. If this attribute is not specified, no limit is used. In such a case, the effective limit is determined by the number of records and the system caching capability.
- SELECT command and JSP syntax

  Represents the only valid SQL command, SELECT. The <tsx:dbquery> tag must return a results set. Refer to your database documentation for information about the SELECT command. See other articles in this section for a description of JSP syntax for variable data and inline Java code.

dbmodify tag JavaServer Pages syntax (deprecated):

The <tsx:dbmodify> tag establishes a connection to a database and then adds records to a database table.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The <tsx:dbmodify> tag does the following:

1. References a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file and uses the information provided by that tag to determine the database URL and driver.

   Note: You can also obtain the user ID and password from the <tsx:dbconnect> tag if those values are provided in the <tsx:dbconnect> tag.
2. Establishes a new connection.
3. Updates a table in the database.
4. Closes the connection and releases the connection resource.

This section describes the syntax of the <tsx:dbmodify> tag.

```
<%-- Any valid database update commands can be placed within the DBMODIFY tag. -->
<%-- Any other syntax, including HTML comments, are not valid. -->
<tsx:dbmodify connection="connection_id">
</tsx:dbmodify>
```

where:

- connection

  Represents the identifier of a <tsx:dbconnect> tag in this JSP file. The <tsx:dbconnect> tag provides the database URL, driver name, and (optionally) the user ID and password for the connection.
- Database commands

  Represents valid database commands. Refer to your database documentation for details

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JavaServer Pages (JSP) file and referenced in the database commands using the <tsx:getProperty> tag.

```
<tsx:dbmodify connection="conn" >
insert into EMPLOYEE
    (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)
values
('<tsx:getProperty name="request" property=request.getParameter("EMPNO") />',
'<tsx:getProperty name="request" property=request.getParameter("FIRSTNME") />',
'<tsx:getProperty name="request" property=request.getParameter("MIDINIT") />',
'<tsx:getProperty name="request" property=request.getParameter("LASTNAME") />',
'<tsx:getProperty name="request" property=request.getParameter("WORKDEPT") />',
<tsx:getProperty name="request" property=request.getParameter("EDLEVEL") />)
</tsx:dbmodify>
```

tsx:getProperty tag JavaServer Pages syntax and examples (deprecated):

The <tsx:getProperty> tag gets the value of a bean to display in a JavaServer Pages (JSP) file.

Note: Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

This IBM extension of the Sun JSP <jsp:getProperty> tag implements all of the <jsp:getProperty> function and adds the ability to introspect a database bean created using the IBM extension <tsx:dbquery> or <tsx:dbmodify>.

Note: You cannot assign the value from this tag to a variable. The value, generated as output from this tag, displays in the browser window.

This section describes the syntax of the <tsx:getProperty> tag:

```
<tsx:getProperty name="bean_name"
  property="property_name" />
```

where:

- **name**

Represents the name of the bean declared by the `id` attribute of a <tsx:dbquery> syntax within the JSP file. Refer to the <tsx:dbquery> article for an explanation. The value of this attribute is case-sensitive.

- **property**

  Represents the property of the bean to access for substitution. The value of the attribute is case-sensitive and is the locale-independent name of the property.

Tag example:

```
<tsx:getProperty name="userProfile" property="username" />
```

tsx:userid and tsx:passwd tag JavaServer Pages syntax (deprecated):

With the <tsx:userid> and <tsx:passwd> tags, you do not have to hard code a user ID and password in the <tsx:dbconnect> tag.

Note: Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

Use the <tsx:userid> and <tsx:passwd> tags to accept user input for the values and then add that data to the request object. You can access the request object with a JavaServer Pages (JSP) file, such as the `JSPEmployee.jsp` example that requests the database connection.

You must use <tsx:userid> and <tsx:passwd> tags within a <tsx:dbconnect> tag.

This section describes the syntax of the <tsx:userid> and <tsx:passwd> tags.

```
<tsx:dbconnect id="connection_id"
    <font color="red"><userid></font>
    <tsx:getProperty name="request" property=request.getParameter("userid") />
    <font color="red"></userid></font>
    <font color="red"><passwd></font>
    <tsx:getProperty name="request" property=request.getParameter("passwd") />
    <font color="red"></passwd></font>
    url="protocol:database_name:database_table"
    driver="JDBC_driver_name">
</tsx:dbconnect>
```

where:

- <tsx:getProperty>

  Represents the syntax as a mechanism for embedding variable data.

- userid

  Represents a reference to the request parameter that contains the user ID. You must add the parameter to the request object that passes to this JSP file. You can set the attribute and its value in the request object, using an HTML form or a URL query string to pass the user-specified request parameters.

- passwd

  Represents a reference to the request parameter that contains the password. Add the parameter to the request object that passes to this JSP file. You can set the attribute and its value in the request object, using an HTML form or a URL query string, to pass user-specified values.

tsx:repeat tag JavaServer Pages syntax (deprecated):

The <tsx:repeat> tag repeats a block of HTML tagging.

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

Use the <tsx:repeat> syntax to iterate over a database query results set. The <tsx:repeat> syntax iterates from the start value to the end value until one of the following conditions is met:
- The end value is reached.
- An exception is thrown.

If an exception of the types **ArrayIndexOutOfBoundsException** or **NoSuchElementException** is created before a block completes, output is written only for the iterations up to and not including the iteration during which the exception was created. All other exceptions results in no output being written for that tag instance.

This section describes the syntax of the <tsx:repeat> tag:

```
<tsx:repeat index="name" start="starting_index" end="ending_index">
</tsx:repeat>
```

where:
- **index**

  Represents an optional name used to identify the index of this repeat block. The scope of the index is NESTED. Its type must be integer.
- **start**

  Represents an optional starting index value for this repeat block. The default is 0.
- **end**

  Represents an optional ending index value for this repeat block. The maximum value is 2,147,483,647.

  If the value of the end attribute is less than the value of the start attribute, the end attribute is ignored.

Combining tsx:repeat and tsx:getProperty JavaServer Pages tags (deprecated)

Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The following code snippet shows you how to code these tags:

```
<tsx:repeat>
<tr>
    <td><tsx:getProperty name="empqs" property="EMPNO" />
    <tsx:getProperty name="empqs" property="FIRSTNME" />
    <tsx:getProperty name="empqs" property="WORKDEPT" />
    <tsx:getProperty name="empqs" property="EDLEVEL" />
    </td>
</tr>
</tsx:repeat>
```

Example: Using tsx:repeat JavaServer Pages tag to iterate over a results set (deprecated):

The <tsx:repeat> tag iterates over a results set. The results set is contained within a bean. The bean can be a static bean, for example, a bean created by using the IBM WebSphere Studio database wizard, or a dynamically generated bean, for example, a bean generated by the <tsx:dbquery> syntax.

Note: Support for tsx tags in the JavaServer Pages (JSP) engine are deprecated in WebSphere Application Server Version 6.0. Instead of using the tsx tags, you should use equivalent tags from the JavaServer Pages Standard Tag Library (JSTL).

The following table is a graphic representation of the contents of a bean called, *myBean*:

Table 119. Graphic representation of myBean.. myBean contents.

Column	col1	col2	col3
row0	friends	Romans	countrymen

Table 119. Graphic representation of myBean. (continued). myBean contents.

Column	col1	col2	col3
row1	bacon	lettuce	tomato
row2	May	June	July

Some observations about the bean:
- The column names in the database table become the property names of the bean. The <tsx:dbquery> section describes a technique for mapping the column names to different property names.
- The bean properties are indexed. For example, `myBean.get(Col1(row2))` returns `May`.
- The query results are in the rows. The <tsx:repeat> tag iterates over the rows, beginning at the start row.

The following table compares using the <tsx:repeat> tag to iterate over a static bean, versus a dynamically generated bean:

Table 120. Static Bean and <tsc:repeat> Bean example. Use this table to compare the two beans.

Static Bean Example	<tsx:repeat> Bean Example

myBean.class
`// Code to get a connection`
 	
`// Code to get the data`
`Select * from myTable;`
 	
`// Code to close the connection`
 	
JSP file
`<tsx:repeat index=abc>`
`<tsx:getProperty name="myBean"`
`property="col1(abc)" />`
`</tsx:repeat>`
 	
Note:
• The bean (myBean.class) is a static bean.
• The method to access the bean properties is myBean.get(*property*(*index*)).
• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.
• The <tsx:repeat> tag iterates over the bean properties row by row, beginning with the start row.
	JSP file
	`<tsx:dbconnect id="conn"`
	`userid="alice"passwd="test"`
	`url="jdbc:db2:sample"`
	`driver="COM.ibm.db2.jdbc.app.DB2Driver">`
	`</tsx:dbconnect >`
	
	`<tsx:dbquery id="dynamic"`
	`connection="conn" >`
	`Select * from myTable;`
	`</tsx:dbquery>`
	
	`<tsx:repeat index=abc>`
	`<tsx:getProperty name="dynamic"`
	`property="col1(abc)" />`
	`</tsx:repeat>`
	
	Note:
	• The bean (dynamic) is generated by the <tsx:dbquery> tag and does not exist until the syntax executes.
	• The method to access the bean properties is dynamic.getValue("*property*", *index*).
	• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.
	• The <tsx:repeat> tag syntax iterates over the bean properties row by row, beginning with the start row.

Implicit and explicit indexing

Examples 1, 2, and 3 show how to use the <tsx:repeat> tag. The examples produce the same output if all indexed properties have 300 or fewer elements. If there are more than 300 elements, Examples 1 and 2 display all elements, while Example 3 shows only the first 300 elements.

Example 1 shows *implicit indexing* with the default start and default end index. The bean with the smallest number of indexed properties restricts the number of times the loop repeats.

```
<table>
<tsx:repeat>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" />
```

```
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="address" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="telephone" />
  </tr></td>
</tsx:repeat>
</table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table>
<tsx:repeat index=myIndex start=0 end=2147483647>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=city(myIndex) />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=address(myIndex) />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=telephone(myIndex) />
</tr></td>
</tsx:repeat>
</table>
```

Example 3 shows *explicit indexing* and ending index with implicit starting index. Although the index attribute is specified, you can still implicitly index the indexed property city because the (`myIndex`) tag is not required.

```
<table>
<tsx:repeat index=myIndex end=299>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" /t>
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="address(myIndex)" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="telephone(myIndex)" />
  </tr></td>
</tsx:repeat>
</table>
```

Nesting <tsx:repeat> blocks

You can nest <tsx:repeat> blocks. Each block is separately indexed. This capability is useful for interleaving properties on two beans, or properties that have subproperties. In the example, two <tsx:repeat> blocks are nested to display the list of songs on each compact disc in the user's shopping cart.

```
<tsx:repeat index=cdindex>
  <h1><tsx:getProperty name="shoppingCart" property=cds.title /></h1>
  <table>
  <tsx:repeat>
    <tr><td><tsx:getProperty name="shoppingCart" property=cds(cdindex).playlist />
    </td></tr>
  </tsx:repeat>
  </table>
 </tsx:repeat>
```

Developing JSF files

Learn about JSF files.

JavaServer Faces

JavaServer Faces (JSF) is a user interface framework or application programming interface (API) that eases the development of Java-based web applications.

WebSphere Application Server supports JavaServer Faces 2.0 at a runtime level. The JSF runtime also:

- Makes it easy to construct a user interface from a set of reusable user interface components
- Simplifies migration of application data to and from the user interface

- Helps manage user interface state across server requests
- Provides a simple model for wiring client-generated events to server-side application code
- Supports custom user interface components to be easily build and reused

Both the Sun Reference Implementation and Apache MyFaces implementation are shipped with the product.

The Apache MyFaces JSF Implementation provides the foundation of the code used for the JSF support in WebSphere Application Server. The version of the JSF runtime provided by the Application Server resides in the normal runtime library location and is available to all web applications that use JSF APIs. Loading the JSF servlet works in the same manner as if the run time was packaged with the web application. The bundled version includes enhancements for better integration with the built-in annotation scanning and other runtime components of WebSphere Application Server.

The specification-related classes (javax.faces.*) for JSF and the IBM modified version of the Apache MyFaces JSF Implementation and the JSF Sun reference implementation are packaged in the Application Server run time.

Typically, web applications that use this API/Framework embed the JSF API and implementation Java archive (JAR) files within their web application archive (WAR) file. This practice is not required when these web applications are deployed and run within WebSphere Application Server. Only the removal of these JAR files along with any JSTL JAR files from the WAR file is required. However, because JavaServer Faces 2.0 is a part of the Java Platform, Enterprise Edition (Java EE) platform, a web application does not need to bundle a JavaServer Faces implementation when it runs on a web container that is Java EE technology compliant. If a JavaServer Faces implementation is bundled with a web application, it is ignored as the JavaServer Faces implementation provided by the platform always takes precedence.

The JSF run time for WebSphere Application Server does not support the use of a single class loader for the entire application. This support is not available when the application contains multiple web modules and one of those modules is a JSF module. A single class loader for the entire application is not supported because the FacesConfig initialization requires a single class loader for each JSF module to perform the initialization. Therefore, you must use multiple class loaders when the application contains multiple web modules and at least one JSF module.

For using different implementations of JSF, the WebSphere Application Server JSF engine determines if the SUN RI or Apache MyFaces is used from the application server run time. After the JSF engine determines the implementation that is used, the correct listener class is registered with the web container. You do not need to add the com.sun.faces.ConfigureListener or the org.apache.myfaces.StartupConfigureListener to the `web.xml` file.

If you want to use a third party JSF implementation that is not shipped with the product, leave the configuration set to MyFaces, add the third party listener to the `web.xml` file that is required and add the third party implementation JAR files to the web module as an isolated shared library. Using an isolated shared library, the web application version of the JSF or JSTL classes load before the Application Server.

JavaServer Faces widget library (JWL)

JavaServer Faces widget library (JWL) is an IBM JSF-based web widget library that integrates widgets from a number of sources. The IBM JSF-based web widget library is deprecated, however, you can obtain the latest version from Rational Application Developer version 6 to work with JSF 1.2.

JWL includes the JSF components from Rational Application Developer except for the base JSF components, which are included in the Application Server run time. This includes the IBM extended JSF components and the extended FacesClient Component. JWL also extends JSF with client-side features for rich browser-based experiences in the form of the FacesClient Component.

Important: JWL is not supported in pages using the JSF 2.0 Facelets format.

Note: You must update the JavaServer Faces widget library (JWL) libraries in JavaServer Faces Web projects that are developed using Rational® Application Developer, Version 7.x with the latest updates from Rational Application Developer before deploying WebSphere® Application Server. Update Rational Application Developer with the latest updates, and update the JWL libraries in the WEB_PROJECT/WEB-INF/lib folder.

JWL Java archive files

JWL is packaged into two Java archive (JAR) files, `odc-jsf.jar` and `jsf-ibm.jar`, which are located in the `${WAS_HOME}\optionalLibraries\IBM\jwl\2.0` directory.

To include JWL in your application, you can use the JWL shared library named JWLLib, which is created at installation time. To assign the library to an application, see the topic, Using installed optional packages.

Configuring Portlet Bridge for JavaServer Faces

Use this task to configure IBM Portlet Bridge for JavaServer Faces (JSF) 2.0.

About this task

Portlet Bridge for JavaServer Faces is available as Version 8.0.0.2.

Procedure

1. Open the portlet.xml file and modify the portlet class for JSF 2.0 portlet bridge, as follows:

```
<portlet-class>com.ibm.faces20.portlet.FacesPortlet</portlet-class>
  <init-param>
    <name>com.ibm.faces.portlet.page.view</name>
    <value>/TestProjectView.xhtml</value>
  <init-param>
```

2. Open the faces-config.xml file and add the JSF 2.0 portlet bridge variable resolver, view handler, and resource handler entry

```
<application>
  <variable-resolver>com.ibm.faces20.portlet.PortletVariableResolver</variable-resolver>
  <view-handler>com.ibm.faces20.portlet.FaceletPortletViewHandler</view-handler>
  <resource-handler>com.ibm.faces20.portlet.httpbridge.PortletResourceHandler</resource-handler>
</application>
```

What to do next

If the portlet project has a custom portlet class, these changes are also necessary:

- Point the portlet-class entry in the portlet.xml file to the particular custom portlet class used.
- Modify the portlet class to extend the com.ibm.faces20.portlet.FacesPortlet class from the IBM Portlet Bridge for JSF 2.0.

Configuring JavaServer Faces implementation

Use this task to specify which JavaServer Faces implementation to use. You can use Apache MyFaces 2.0 or the SUN Reference Implementation 1.2 of JSF, or your own implementation.

Before you begin

Ensure that your application is configured for JavaServer Faces (JSF) using the specific `web.xml` context parameters for the implementation that you have chosen.

Attention: The JSF implementation is a server-wide configuration setting. Thus, if you have multiple applications which require different JSF implementations, you must separate the applications into different application servers or clusters, and specify the implementation by following the steps in this topic for each application server or cluster. Alternatively, you can use an isolated shared library, as described for third-party JSF implementations, as follows.

About this task

Note: The Application Server JSF engine determines if the SUN Reference Implementation (RI) 1.2 or Apache MyFaces 2.0 is used from the Application Server run time. If either is used, the correct listener class is registered with the web container. You do not need to add the com.sun.faces.ConfigureListener or the org.apache.myfaces.StartupConfigureListener to your `web.xml` file.

Be Aware: If you want to use a third-party JSF implementation that is not shipped with the product, then:

- Keep the configuration set to MyFaces.
- Add the third-party listener to the `web.xml` file that is required.
- Add the third-party implementation Java archive (JAR) files to the application as an isolated shared library and associate it with your application.

You can also configure the JSF implementation on the Provide JSP reloading options for web modules panel for application installation and update wizards.

Procedure

Configure the server or cluster to use the JSF implementation that you want. You can do this task using the administrative console or the wsadmin tool.

- In the administrative console panel, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **JSP and JSF options**

  Select one of the following implementations:
  - Sun Reference Implementation 1.2 - Select this option to use the Sun Reference Implementation 1.2 JSF implementation.
  - MyFaces 2.0 - Select this option to use the MyFaces 2.0 JSF implementation. This option is the default JSF implementation.
- Using the wsadmin tool:
  - An example of setting a single server to use the Sun RI 1.2 JSF implementation:

```
wsadmin>set server [$AdminConfig list ApplicationServer *server1*]
server1(cells/myNode01Cell/nodes/myNode01/servers/server1|server.xml#ApplicationServer_1183122130078)
wsadmin>$AdminConfig modify $server {{jsfProvider SunRi1.2}}

wsadmin>$AdminConfig save
```

  - An example of setting a cluster to use the MyFaces 2.0 JSF implementation:

```
wsadmin>set cluster [$AdminConfig list ServerCluster]
cluster1(cells/myNode01Cell/nodes/myNode01/clusters/cluster1|cluster.xml#ServerCluster_1173916133721)
wsadmin>$AdminConfig modify $cluster {{jsfProvider MyFaces}}

wsadmin>$AdminConfig save
```

Results

What to do next

Configure JSF engine parameters as necessary.

Configuring JSF engine parameters:
About this task

WebSphere Application Server does not support the modification of deployment descriptor extension parameters through the administrative console or through administrative scripting.

To add, change or delete JSF engine configuration parameters, complete the following steps:

Procedure

1. Open the WEB-INF/web.xml file.

   JSP engine configuration parameters are stored in a web module's configuration directory or in a web modules's binaries directory in the WEB-INF/web.xml file. Open the WEB-INF/web.xml file from:

   - The configuration directory, as in the following example: {WAS_ROOT}/profiles/*profilename*/config/cells/*cellname*/applications/*enterpriseappname*/ deployments/*deployedname*/*webmodulename*
   - The binaries directory if an application was deployed into WebSphere Application Server with the flag "Use Binary Configuration" set to true. An example of a binaries directory is: {WAS_ROOT}/profiles/*profilename*/installedApps/*nodename*/*EnterpriseAppName*/*WebModuleName*/

2. Edit the WEB-INF/web.xml file.

   - To add configuration parameters, use the following format:

     ```
     <context-param>
       <description>descriptive text</description>
       <param-name>parameter name</param-name>
       <param-value>parameter value</param-value>
     </context-param>
     ```

   - To delete configuration parameters, either delete the line from the file, or enclose the statement with <!-- --> tags.

3. Save the file.

4. Restart the Enterprise Application. It is not necessary to restart the server for parameter changes to take effect. However, some JSP engine configuration parameters affect the Java source code that is generated for a JSP. If such a parameter is changed, then you must retranslate the JSP files in the web module to regenerate Java source. You can use the batch compiler to retranslate all JSP files in a web module. The batch compiler uses the JSP engine configuration parameters that you have set in the web.xml file, unless you specifically override them. The JSP engine configuration topic identifies the parameters that affect the generated Java source.

Example

The following is a sample of the WEB-INF/web.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 id="WebApp_ID" version="2.5">
 <display-name>Servlet 2.5 example</display-name>
 <welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.htm</welcome-file>
  <welcome-file>default.jsp</welcome-file>
 </welcome-file-list>
 <context-param>
       <param-name>javax.faces.CONFIG_FILES</param-name>
       <param-value>/WEB-INF/faces-config.xml</param-value>
    </context-param>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>server</param-value>
    </context-param>
    <context-param>
        <param-name>com.ibm.ws.jsf.LOAD_FACES_CONFIG_AT_STARTUP</param-name>
        <param-value>true</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
```

```
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
</web-app>
```

JSF engine configuration parameters:

In WebSphere Application Server, you can configure the JavaServer Faces (JSF) engine configuration parameters for optimal performance in a production server environment and for the needs of developers in a development environment.

The JSF engine parameters are case-sensitive. If the value specified for a parameter is composed of two or more words separated by spaces, you must add quotation marks around the value.

JSF options using SUN RI

Table 121. JSF options using SUN RI. The table shows SUN RI parameter names, descriptions and default values.

SUN RI parameter name	Description	Default value
com.sun.faces.numberOfViewsInSession	Specifies the number of views that are stored in the session when Server-Side State Saving is used. If set to true while client-side state saving is being used, reduces the number of bytes sent to the client by compressing the state before it is encoded and written as a hidden field.	15
com.sun.faces.numberOfLogicalViews	Specifies the number of logical views that are stored in the session when Server-Side State Saving is used.	15
com.sun.faces.enableHighAvailability	If set to true while server-side state saving is used, a serialized representation of the view is stored on the server. This provides failover and server clustering support.	false
com.sun.faces.injectionProvider	Defines an injection provider that is used for JSF annotations.	
com.sun.faces.serializationProvider	Defines a serialization provider that is used for serializing JSF objects into session.	
com.sun.faces.responseBufferSize	Define the size of the response buffer for a JSF response.	1048
com.sun.faces.clientStateWriteBufferSize		8192
com.sun.faces.expressionFactory	Specifies the default EL Expression Factory to use.	org.apache.el.ExpressionFactoryImpl
com.sun.faces.clientStateTimeout	The timeout value used for client side state saving. When the value set has been reached then the state is lost.	infinite
com.sun.faces.displayConfiguration		false
com.sun.faces.validateXml	When set to true, tag library XML files and faces configuration XML files using schema are validated during application start.	false
com.sun.faces.verifyObjects		false
com.sun.faces.forceLoadConfiguration		false
com.sun.faces.disableVersionTracking		false
com.sun.faces.enableHtmTagLibValidator		false
com.sun.faces.prerefXHTML		false
com.sun.faces.compressViewState		true
com.sun.faces.compressJavaScript		true
com.sun.faces.sendPoweredByHeader		true
com.sun.faces.enableJSStyleHiding		false
com.sun.faces.writeStateAtFormEnd		true
com.sun.faces.enableLazyBeanValidation		true
com.sun.faces.enableLoadBundle11Compatibility		false
com.sun.faces.enableRestoreView11Compatibilty		false
com.sun.face.serializeServerState		false
com.ibm.ws.jsf.JSP_UPDATE_CHECK	This parameter monitors Faces JavaServer Pages (JSP) files for modifications and synchronizes a running server with the changes without restarting the server. If this parameter is set to false or removed from the deployment descriptor, any changes made to Faces JSP files might not be seen by the server until it is restarted. Set this parameter to true while developing and debugging the Faces JSP files to improve the performance of the development environment.	

Table 121. JSF options using SUN RI (continued). The table shows SUN RI parameter names, descriptions and default values.

SUN RI parameter name	Description	Default value
com.ibm.ws.jsf.JSF_IMPL_CHECK	Set the com.ibm.ws.jsf.JSF_IMPL_CHECK parameter to true to check at application restart if the SUN RI and MyFaces implementations were switched. If the implementation has switched, then the runtime removes any generated JSP files from the temp directory and the JSP file is retranslated the next time it is requested.	

com.ibm.ws.jsf.associateLabelWithId	The com.ibm.ws.jsf.associateLabelWithId custom property changes the rendering behavior for both the <h:selectOneRadio> and <h:selectManyCheckbox> components. The label no longer wraps the input element. Instead, each input element has a unique ID and the label is associated with that ID used for that attribute. Define and set the com.ibm.ws.jsf.associateLabelWithId context parameter to `true` in the `web.xml` file.
	
	Use the following code as an example.
	`<context-param>`
	`<description>`
	`Set to true to explicitly associate labels`
	`with their input elements for select one`
	`radio buttons and select many check box lists.`
	`</description>`
	`<param-name>com.ibm.ws.jsf.associateLabelWithId`
	`</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`	

com.ibm.ws.jsf.disableEnqueuedMessagesWarning	The com.ibm.ws.jsf.disableEnqueuedMessagesWarning custom property disables the `FacesMessage(s) have been enqueued, but may not have been displayed` warning message. When this property is set to `true` in the `web.xml` file, this warning message is not included in the `SystemOut.log` file .
	gotcha: This context parameter only applies to JSF applications that use the Sun Reference Implementation (RI) for JSF implementation.
	Note: This topic references one or more of the application server log files. As a recommended alternative, you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files on distributed and IBM i systems. You can also use HPEL in conjunction with your native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.
	
	Use the following code as an example of how to define and set the com.ibm.ws.jsf.disableEnqueuedMessagesWarning context parameter to `true` in the web.xml file.
	`<context-param>`
	`<description>`
	`Set to true to disable the following warning`
	`message:`
	`FacesMessage(s) have been enqueued, but might`
	`not have been displayed`
	`</description>`
	`<param-name>`
	` com.ibm.ws.jsf.disableEnqueuedMessagesWarning`
	`</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`	

com.ibm.ws.jsf.disableStylePassthroughForCheckboxList	This custom property prevents passing the style information into the items in the check box list. This property defaults to false to maintain the current behavior. Define and set the com.ibm.ws.jsf.disableStylePassthroughForCheckboxList context parameter to `true` in the web.xml file prevent passing style information into items in the check box list. Use the following code as an example.
	`<context-param>`
	`<description>`
	`Set to true if style information should not`
	`be passed into items of check box list`
	`</description>`
	`<param-name>`
	`com.ibm.ws.jsf.disableStylePassthroughForCheckboxList`
	`</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`	

Table 121. JSF options using SUN RI (continued). The table shows SUN RI parameter names, descriptions and default values.

SUN RI parameter name	Description	Default value

com.ibm.ws.jsf.DISABLE_UIDATA_NESTED_CHECK	The com.ibm.ws.jsf.DISABLE_UIDATA_NESTED_CHECK custom property determines whether unique IDs are generated for UIData components that are nested inside iterator components that are not UIData components. When this property is set to `true` in the web.xml file, unique IDs are generated for UIData components even if they are nested inside iterator components that are not UIData components.
	
	When this property is set to `false`, if a JSF dataTable component is nested within a component that does not extend UIData, such as the Java Widget Library (JWL) dataIterator component, the IDs that are generated for the rows do not increment properly. This situation can result in duplicate IDs being assigned to multiple JSF components.
	
	Use the following code as an example of how to define and set the com.ibm.ws.jsf.DISABLE_UIDATA_NESTED_CHECK context parameter to `true` in the `web.xml`file.
	`<context-param>`
	`<description>`
	`Set to true to enable unique IDs to be`
	`generated for UIData components evenif they`
	`are nested inside iterator components that are`
	`not UIData components.`
	`</description>`
	`<param-name>`
	`com.ibm.ws.jsf.DISABLE_UIDATA_NESTED_CHECK`
	`</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`	

com.ibm.ws.jsf.loadExternalDtd	When parsing the faces-config.xml file from included libraries, the Faces configuration parser attempts to load the DTD even when validation is disabled. The Faces configuration parser uses a SAXParser to read the faces-config.xml. The default behavior of the SAXParser parser is to always load the DTD even if validation is disabled. This behavior can lead to errors initializing the Faces Servlet on systems isolated from the internet.
	
	In your `web.xml` file, set the com.ibm.ws.jsf.loadExternalDtd context paramater to `false` to have the Faces configuration parser set the "http://apache.org/xml/features/nonvalidating/load-external-dtd" feature to false.
	`<context-param>`
	`<description>`
	`When set to false, this property sets a`
	`feature on the SAX parser to prevent loading`
	`the external DTD.`
	`</description>`
	`<param-name>com.ibm.ws.jsf.loadExternalDtd</param-name>`
	`<param-value>false</param-value>`
	`</context-param>`	

enableRestoreView11Compatibility	A JSF 1.2 application might create the ViewExpiredException exception under load when using the Sun Reference Implementation. If your view is not found in session, you can use a compatibility mode in JSF to create a new view. This can have adverse behaviors because it is a new view and items that are usually in the view, such as state, will no longer be available. Use the following code as an example to set the com.sun.faces.enableRestoreView11Compatibility context parameter to `true` in the web.xml file. This is only applicable when the Sun Reference Implementation is in use.
	`<context-param>`
	`<param-name>`
	`com.sun.faces.enableRestoreView11Compatibility`
	`</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`	

enableViewStateIdRendering	The com.sun.faces.enableViewStateIdRendering custom property controls the rendering of the id attribute of the javax.faces.ViewState hidden field. Use the following code as an example to set the com.sun.faces.enableViewStateIdRendering context parameter to `true` in the web.xml file.
	`<context-param>`
	`<param-name>com.sun.faces.enableViewStateIdRendering`
	`</param-name>`
	`<param-value>true</param-value>`
	`</context-param>`	

JSF options for MyFaces

Table 122. JSF options for MyFaces. The table shows JSF parameter names, descriptions and default values.

JSF parameter name	Description	Default value
org.apache.myfaces.RESOURCE_VIRTUAL_PATH		/faces/myFacesExtensionResource

Table 122. JSF options for MyFaces (continued). The table shows JSF parameter names, descriptions and default values.

JSF parameter name	Description	Default value
org.apache.myfaces.PRETTY_HTML	If this value is true, rendered HTML code is formatted so that it is human-readable. Additional line separators and white space are written that do not influence the HTML code.	true
org.apache.myfaces.ALLOW_JAVASCRIPT	This parameter tells MyFaces if javascript code is allowed in the rendered HTML output. If javascript is allowed, command_link anchors have javascript code that submits the corresponding form. If javascript is not allowed, the state saving information and nested parameters are added as URL parameters.	true
org.apache.myfaces.DETECT_JAVASCRIPT		false
org.apache.myfaces.AUTO_SCROLL	If true, a javascript function is rendered that can restore the former vertical scroll on every request. This feature is convenient if you have pages with long lists and you do not want the browser page to jump to the top of the page if you trigger a link or button action that stays on the same page.	false
org.apache.myfaces.ADD_RESOURCE_CLASS		org.apache.myfaces.renderkit.html.util.DefaultAddResource
org.apache.myfaces.CHECK_EXTENSIONS_FILTER	This parameter checks for a properly-configured Extensions-Filter if it is needed by the web application.	true
org.apache.myfaces.COMPRESS_STATE_IN_SESSION	Set this option to `true` to compress the serialized state before it is written to the session. If this option is set to `false`, the state is not compressed. This option is only applicable if the state saving method is set to server and if org.apache.myfaces.SERIALIZE_STATE_IN _SESSION is set to `true`.	true
org.apache.myfaces.NUMBER_OF_VIEWS_IN_SESSION	Defines the number of the latest views that are stored in session. This option is only applicable if the state saving method is set to server.	20
org.apache.myfaces.READONLY_AS_DISABLED_FOR _SELECTS		true
org.apache.myfaces.SERIALIZE_STATE_IN_SESSION	Set this option to `true` to serialize the state to a byte stream before it is written to the session. If this option is set to `false`, the state is not serialized to a byte stream. This option is only applicable if the state saving method is set to server.	true
org.apache.myfaces.STRICT_JSF_2_CC_EL_RESOLVER	Ensures that, when a getType() is called over the source EL expression, components working with chained EL expressions can use the metadata information that composite:attribute added. Setting this property to `true` disables this function.	false
com.ibm.ws.jsf.disablealternatefacesconfigsearch	Disables MyFaces searching for META-INF/*.faces-config.xml for only the web application that this context parameter is set on. If the context parameter and the web container custom property are set, the context parameter takes precedence.	false

Table 122. JSF options for MyFaces (continued). The table shows JSF parameter names, descriptions and default values.

JSF parameter name	Description	Default value

org.apache.el.parser.COERCE_TO_ZERO	Allows for the expression language (EL) that WebSphere Application Server uses to coerce null and empty string integer values to a 0 value or for NOT allowing a coerce to a 0 value and retaining the null or empty string integer. The default is `true` and permits a null or empty string integer value to be coerced to a 0 value.
	Important: In order to keep a null value from being coerced to a 0 value in a MyFaces application, the following context parameter in the `web.xml` of the application: should be set to ensure that all possible instances of an empty or null value are inhibited from being coerced to zero.
	`<context-param>`
	` <param-name>javax.faces.`
	`  INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL`
	` </param-name>`
	` <param-value>true</param-value>`
	`</context-param>`
	
	You set the `org.apache.el.parser.COERCE_TO_ZERO` property using the administrative console.
	1. Expand Servers > Select WebSphere Application Servers > Click on the appropriate server from the list.
	2. Under Server Infrastructure, expand Java and Process Management > Click on Process definition.
	3. Under Additional Properties, click Java virtual Machine.
	4. Under Additional Properties, click Custom properties.
	5. Click `New` and add the `org.apache.el.parser.COERCE_TO_ZERO` property with the value of `false` if you do NOT want a null value coerced to zero.
	6. Click `Save` to save the change and restart the WebSphere Application Server to ensure the change takes place.
		true

org.apache.myfaces.DEBUG_PHASE_LISTENER	Enables the `DebugPhaseListener` in the Development Project Stage.	true

JSF options using SUN RI or MyFaces

The following options are valid for both the SUN RI and the MyFaces implementations.

Table 123. JSF options using SUN RI or MyFaces. The table shows JSF parameter names, descriptions and default values.

JSF parameter name	Description	Default value
javax.faces.STATE_SAVING_METHOD	Specifies the location where state information is saved. Valid values are 'server', which is saved in HttpSession, and 'client', which is saved as a hidden field in the form.	server
javax.faces.CONFIG_FILES	Use this parameter to specify a comma-delimited list of context-relative resource paths under which the JSF implementation looks for application configuration resources before loading a configuration resource named `/WEB-INF/facesconfig.xml`, if a resource exists.	n/a
javax.faces.DEFAULT_SUFFIX	Specifies the default suffix for extension-mapped resources that contain JSF components.	.jsp
javax.faces.LIFECYCLE_ID	Use this parameter to configure an alternate life cycle ID.	n/a

Table 123. JSF options using SUN RI or MyFaces (continued). The table shows JSF parameter names, descriptions and default values.

JSF parameter name	Description	Default value
com.ibm.ws.jsf.JSF_IMPL_CHECK	Specifies that the JSP files in a web module must be recompiled when the application is restarted because the implementation of JSF that is used has changed. After the application is restarted, the next time a JSP file is accessed for this module the JSP file is recompiled against the selected implementation of JSF specified in the administration console. Subsequent calls to the JSP file do not recompile. The default setting for this option is false. Use this option for development and not in a production environment.	n/a

Sun RI context parameters that have an equivalent behavior in MyFaces

Table 124. Sun RI context parameters and equivalent MyFaces behavior. The table shows SUN RI parameters names and MyFaces equivalent.

SUN RI parameter name	Description	RI default	MyFaces equivalent	MyFaces default
com.sun.faces.numberOfViewsInSession	Defines the maximum number of serialized views stored in the session. Works with server state saving.	15	org.apache.myfaces.NUMBER_OF_VIEWS_IN_SESSION	20
com.sun.faces.compressViewState	When true the view is compressed after it is serialized and before base64 encoded. Works with client state saving. As of 1.2_09, this option also affects server-side state saving when com.sun.faces.serializeServerState is set to true (this has a large impact of the size of the state in the session when using this option, at the expense of more CPU.)	true	org.apache.myfaces.COMPRESS_STATE_IN_CLIENT for client-side state saving or org.apache.myfaces.COMPRESS_STATE_IN_SESSION for server-side state saving	false for client-side state saving, true for server-side state saving
com.sun.faces.validateXml	When true JSF validates the configuration files.	false	org.apache.myfaces.VALIDATE	false
com.sun.faces.injectionProvider	This parameter specifies a class that implements the InjectionProvider.	n/a	injection provider is provided by the WebSphere Application Server run time	n/a
com.sun.faces.serializationProvider	This parameter specifies a class that implements the SerializationProvider SPI. This implementation represents a hook the JSF implementation uses to enable using alternate Serialization implementations.	n/a	org.apache.myfaces.SERIAL_FACTORY - class must implement org.apache.myfaces.shared_impl.util.serial.SerialFactory SPI instead of com.sun.faces.spi.SerializationProvider	n/a
com.sun.faces.enabledJSStyleHiding	If true, inline JavaScript rendered by the HTML ResponseWriter implementation is rendered so that the script is hidden from older browser implementations.	false	org.apache.myfaces.WRAP_SCRIPT_CONTENT_WITH_XML _COMMENT_TAG	true
com.sun.faces.serializeServerState	If enabled the component state (not the tree) is serialized before being stored in the session. This might be desirable for applications that have issues with view state being sensitive to model changes.	false	org.apache.myfaces.SERIALIZE_STATE_IN_SESSION	true

Defining an extension for the registry filter

The registry filter specifies if an extensions is applicable to all registry instances or to specified instances.

Before you begin

You must have an extensible application to define an extension for the registry filter.

About this task

Complete the following steps to filter out extensions for an application.

Procedure

1. Define an extension for the registry filter extension point for a named registry instance in the `plugin.xml` file.

```
<extension point="org.eclipse.extensionregistry.RegistryFilter">
 <filter name="AdminConsole*"
   class="com.ibm.ws.admin.AdminConsoleExtensionFilter"/>
 </extension>
```

2. Add the filter implementation to the application by creating a class to implement the com.ibm.workplace.extension.IExtensionRegistryFilter interface.

```
package com.ibm.ws.admin;
import com.ibm.workplace.extension.IExtensionRegistryFilter;
public class AdminConsoleExtensionFilter implements IExtensionRegistryFilter {
        :
}
```

3. The extensible application declares the registry name by defining an extension for the RegistryInstance extension point. This way, the registry can prepare an IExtensionRegistry instance and put it in JNDI in advance.

```
<extension point="org.eclipse.extensionregistry.RegistryInstance">
  <registry name="AdminConsole"/>

</extension>
```

4. The extensible application obtains a named instance of the registry to activate any associated filters:

```
InitialContext ic = new InitialContext();
String lookupName = "services/extensionregistry/AdminConsole";
IExtensionRegistry reg = (IExtensionRegistry)ic.lookup( lookupName );
```

Application extension registry

WebSphere Application Server has enabled the Eclipse extension framework for applications to use. Applications are extensible when they contain a defined extension point and provide the extension processing code for the extensible area of the application.

An application can be plugged in to another extensible application by defining an extension that adheres to what the target extension point requires. The extension point can find the newly added extension dynamically and the new function is seamlessly integrated in the existing application. It works on a cross Java Platform, Enterprise Edition (Java EE) module basis. The application extension registry uses the Eclipse plug-in descriptor format and application programming interfaces (APIs) as the standard extensibility mechanism for WebSphere applications. Developers that build WebSphere application modules can use WebSphere Application Server extensions to implement their functionality to an extensible application, which defines an extension point. This is done through the application extension registry mechanism.

The architecture of extensible Java EE applications follow a modular design to add new functional modules or to replace an existing module, particularly by those outside of its core development team. Each module is a pluggable unit, or plug-in that is either deployed into the portal or removed from the Java EE application using a deployment tool that is based upon standard Java EE and portal web module deployment tooling. A plug-in module describes where it is extensible and what capability it provides to other plug-ins in the `plugin.xml` file. The `plugin.xml` manifest file can be created with a simple text editor or in Eclipse's Plug-in Development Environment (PDE), which provides a simplified view of the same underlying XML data.

WebSphere Application Server implementations to the Eclipse model

Some minor differences exist in the WebSphere Application Server implementation of this architecture because of platforms, specifically, Eclipse Workbench or Java 2 Platform, Enterprise Edition (Java EE). The highlights of the WebSphere Application Server implementation include:

- Implementing all of the extension registry-related interfaces from Eclipse 3.6 and later.
- The identical `plugin.xml` syntax, however, some attributes are not used, for example, <runtime>.
- The discovery and addition of plug-ins to the registry, when the containing Java EE module starts, and plug-ins are dismissed and removed from the registry when the containing Java EE module stops.
- Access to an IExtensionRegistry object is through the Java Naming and Directory Interface (JNDI), instead of by using the Platform.getExtensionRegistry method in the Eclipse Workbench.
- Filtering capability is available by providing a filter implementation and using a named registry instance that finds and invokes the filter as necessary. See the developer API documentation for the IExtensionRegistryFilter interface for more details.

Available Eclipse 3.6 interfaces

The following Eclipse 3.6 and later interfaces are available on WebSphere Application Server:

- Extension registry API
- Extension point API
- Extension API
- Configuration element API
- Registry change listener API
- Registry change event API
- Extension delta API
- Status API

The following interfaces are recognized and processed the same as in Eclipse:

- Executable extension API
- Executable extension factory API

Application extension registry filtering

The extension registry exposes the registry filter extension point. The registry filter removes elements within the extension registry for client applications. Extensions that are attached to the registry filter extension point and that also implement this interface are called as necessary when a client operates on a named registry instance that matches the target specification.

You can create a filter extension for all registry instances or for named instances that are specified by the extension. In the first case, the filter is applied to all instances of the extension registry, and all client applications use the filter without requesting the filter. In the latter case, a client application must predefine the registry name by defining an extension, called *RegistryInstance*, which is another extension point that is exposed by the extension registry. After the registry name is defined, the client can obtain the named registry instance and use that registry instance. The filter extension is invoked by the named registry instance as necessary.

Registry filter API

Supported arguments include:

`org.eclipse.core.runtime.IExtension[]`
doFilter(org.eclipse.core.runtime.IExtension[] extensions)

This code returns an array of IExtension objects that are included in the valid extension list.

Registry instance extension point

The extension registry exposes the *RegistryInstance* . The instance name is declared in the application's `plugin.xml` file, and the application requests an registry instance for that name at runtime.

plugin.xml file

A plug-in is described in an XML manifest file, called `plugin.xml`, which is part of the plug-in deployment files. The manifest file tells the portal application's runtime what it needs to know to register and activate the plug-in. The manifest file essentially serves as the contract between the pluggable component and the portal application's runtime. Although the WebSphere Application Server `plugin.xml` closely follows the one provided for the Eclipse workbench, it does diverge from the Eclipse workbench in several places as outlined below.

Location

The `plugin.xml` file must reside in the `WEB-INF` directory under the context of the hierarchy of directories that exist for a web application or when included in the Web application archive file.. The `plugin.xml` file must reside in the root directory when the`plugin.xml` file is placed in an Enterprise JavaBeans Java archive (JAR) file or shared library JAR file. The extension registry service includes the `plugin.xml` file as the participating components are loaded and started on the application server.

Usage notes

- Is this file read-only?

  No
- Is this file updated by a product component?

  ???
- If so, what triggers its update?

  Rational Application Developer updates the `web.xml` file when you assemble web components into a web module, or when you modify the properties of the web components or the web module.
- How and when are the contents of this file used?

  WebSphere Application Server functions use information in this file during the configuration and deployment phases of web application development.
- The manifest markup definitions below make use of various naming tokens and identifiers. To eliminate ambiguity, the following are productions rules for these naming conventions. In general, all identifiers are case-sensitive.

```
SimpleToken := sequence of characters from ('a-z','A-Z','0-9')
ComposedToken := SimpleToken | (SimpleToken '.' ComposedToken)
PlugInId := ComposedToken
PlugInPrereq := PlugInId
ExtensionId := SimpleToken
ExtensionPointId := SimpleToken
ExtensionPointReference := ExtensionPointId | (PlugInId '.' ExtensionPointId)
```

Sample file entry

The entire plug-in manifest DTD is as follows. XML Schema is not used to define the manifest since the current Eclipse tooling for plug-in's requires a DTD. The XML DTD construction rule element* means zero or more occurrences of the element; element? means zero or one occurrence of the element; and element+ means one or more occurrences of the element.

```
<?xml encoding="US-ASCII"?>

<!ELEMENT plugin (requires?, extension-point*, extension*)>
<!ATTLIST plugin
  name CDATA #IMPLIED
  id CDATA #REQUIRED
  version CDATA #REQUIRED
  provider-name CDATA #IMPLIED
```

```
>
<!ELEMENT requires (import+)>
<!ELEMENT import EMPTY>
<!ATTLIST import
  plugin CDATA #REQUIRED
  version CDATA #IMPLIED
  match (exact | compatible | greaterOrEqual) #IMPLIED
>
<!ELEMENT extension-point EMPTY>
<!ATTLIST extension-point
  name CDATA #IMPLIED
  id CDATA #REQUIRED
  schema CDATA #IMPLIED
>
<!ELEMENT extension ANY>
<!ATTLIST extension
  point CDATA #REQUIRED
  id CDATA #IMPLIED
  name CDATA #IMPLIED
>
```

WebSphere Application Server differences

The `plugin.xml` file closely follows the `plugin.xml` file provided for the Eclipse workbench. However it diverges within the following elements.

The `plugin` element

The plugin element provided in this manifest does not contain class attributes. The class attribute is unnecessary since the plug-in mechanism does not require the plug-in developer to extend or use any specific classes as is required by the Eclipse workbench. Also, the plugin element does not contain a runtime element since standards such as J2EE that already define the location of runtime libraries for the applications.

The `import` element

The requires element does not contain export attribute since J2EE modules are encouraged to be self-contained to improve manageability. In addition to eliminating the export attribute, the match attribute has an option for a greater than or equal to match for versions (greaterOrEqual).

The `extension-point` element

The extension-point element has the name attribute as optional since it has no real use in this J2EE implementation.

you can find details regarding the plug-in manifest in the Eclipse documentation, under Platform Plug-In Developer Guide>Other reference information>Plug-in manifest.

The following is an example of how adding a link to an existing page can be accomplished by an extension point. The plug-in manifest of this plug-in declares an extension point (linkExtensionPoint) and an extension to this extension point (linkExtension). The plug-in declaring the extension point does not need to be the plug-in that implements the extension point. Another plug-in can also define an extension to the link extension point in its plug-in manifest by including the contents of the <extension> and </extension> tags in its manifest.

```
<?xml version="1.0"?>
<!--the plugin id is derived from the vendor domain name -->
<plugin
   id="com.ibm.ws.console.core"
   version="1.0.0"
   provider-name="IBM WebSphere">

  <!--declaration of prerequisite plugins-->
 <requires>
    <import plugin="com.ibm.data" version="2.0.1" match="compatible"/>
    <import plugin="com.ibm.resources" version="3.0" match="exact"/>
```

```
 </requires>

 <!--declaration of link extension point -->
 <extension-point
    id="linkExtensionPoint"
    schema="/schemas/linkSchema.xsd"/>


 <!--declaration of an extension to the link extension point -->
 <extension
    point="com.ibm.ws.console.core.linkExtensionPoint"
    id="linkExtension">

   <link
     label="Example.displayName"
     actionView="com.ibm.ws.console.servermanagement.forwardCmd.do?
        forwardName=example.config.view&
        lastPage=ApplicationServer.config.view">
   </link>
 </extension>
</plugin>
```

Contexts and Dependency Injection (CDI)

Learn about Contexts and Dependency Injection (CDI).

Developing applications that use Contexts and Dependency Injection (CDI)

Use this task to provide a summary of the WebSphere Application Server extensions that you can use to develop servlets.

About this task

Several WebSphere Application Server extensions are provided for enhancing your servlets. This task provides a summary of the extensions that you can use.

Procedure

1. Review the supported specifications.

   Create Java™ components, referring to the CDI specifications. Place a beans.xml file in the WEB-INF directory of the WAR module, or META-INF directory of a JAR file, so the container identifies it as a bean deployment archive.
2. Use your favorite integrated development environment (IDE), or a text editor, to develop or migrate code artifacts that meet the specifications.
3. Test the code artifacts.

What to do next

Assemble your code artifacts into a web module using assembly tools as a prerequisite to deploying the code to the application server.

Contexts and Dependency Injection (CDI)

Contexts and Dependency Injection for the Java EE platform (CDI) is a JSR 299 implementation that is based on Apache OpenWebBeans.

CDI is activated in an application by the presence of a beans.xml file inside that module, as defined by the JSR 299 specification. You can find the beans.xml file in the WEB-INF directory of a web archive (WAR) or META-INF directory of other types of archives. When activated, the container provides services such as:

- Context management
- Type-safe dependency injection: A CDI-managed bean is instantiated and injected as needed.

- Decorators, which implement one or more bean interfaces and can contain business logic. Decorators are disabled by default. You can have multiple decorators per bean and order is defined by the beans.xml file.
- Interceptor bindings. Interceptors, which are enabled manually in the beans.xml file, are bound using an interceptor binding type.
- Event model
- Integration into JavaServer Faces (JSF) and JavaServer Pages (JSP) files using the Expression Language (EL)

The specification-related API classes for JSR 299 and JSR 330 and IBM modified implementation classes that are based on Apache OpenWebBeans are packaged with the application server runtime environment.

Although the WebSphere Application Server CDI implementation is based on Apache OpenWebBeans, there are some changes and additions on top of OpenWebBeans to support integration with the server run time:

- Integration with other Java Platform, Enterprise Edition (Java EE) containers in WebSphere Application Server that support injectable components.
- ScannerService implementation that uses the WebSphere Application Server byte code scanner.
- Direct use of WebSphere Application Server Enterprise JavaBeans (EJB) metadata for determining EJB types.
- Automatic registration of Servlet Listeners, Filters, Interceptors for CDI applications so these no longer must be added by each application.
- WebSphere Application Server-specific implementations of many OpenWebBeans Service Programming Interfaces (SPI), such as ResourceInjectionService, TransactionService, failover service, and so on.

Important: Container-managed transactions and security are not provided by CDI.

Contexts and Dependency Injection custom properties:

WebSphere Application Server Contexts and Dependency Injection (CDI) implementation is based on OpenWebBeans, which is configurable through the openwebbeans.properties file.

To specify CDI custom properties, place the openwebbeans.properties file inside the META-INF/openwebbeans directory of your application. Properties that are not listed below, such as those that control the lifecycle and services, should remain at the default value for the application server. See the following list of some of the available CDI custom properties:

org.apache.webbeans.conversation.Conversation.periodicDelay:

Specifies a delay in milliseconds. Use the conversation periodic delay custom property to search for and delete unused conversations.

Value	Default
integer	15000

org.apache.webbeans.application.jsp:

Specifies that an application is a full JavaServer Pages (JSP) application. Specify true if you want to add the JSP ELResolver class to the application.

Value	Default
true	true

org.apache.webbeans.useOwbSpecificXmlConfig:

Specify true if you want to use OpenWebBeans-specific beans.xml files for the application. The default value is false, which is the portable specification beans.xml format.

Value	Default
true	false

org.apache.webbeans.fieldInjection.useOwbSpecificInjection:

Specify true if you want to use OpenWebBeans-specific injection for the application. The default value is false, which is the portable specification behavior.

Value	Default
true	false

org.apache.webbeans.application.useJSF2Extensions:

Specify true to enable JavaServerFaces (JSF) 2.0-based CDI extensions.

Value	Default
true	false

org.apache.webbeans.application.supportsConversation:

Specify true to support conversation scopes in the application.

Value	Default
true	true

CDI integration with JavaServer Faces:

Contexts and Dependency Injection (CDI) primarily integrates with JavaServer Faces (JSF) through the Expression Language (EL). It enables CDI beans to be exposed through the unified EL-to-JSF components. It also provides a built-in context for conversation scope that is active during standard JSF life cycle phases.

As part of the WebSphere Application Server integration with CDI containers, several JSF components are automatically registered for CDI applications, including:

- EL Resolver
- Phase Listener
- View Handler
- Application Factory

Only the default JSF implementation (based on MyFaces) is supported for use with CDI.

Contexts and Dependency Injection (CDI) integration with EJB container:

The CDI specification enhances the EJB component model with contextual life cycle management.

Relationship of the CDI to the EJB specification

The EJB specification defines a programming model for application components that access transactional resources in a multi-user environment. Concerns, such as role-based security, transaction demarcation, concurrency, and scalability are specified declaratively using annotations and XML deployment descriptors that are enforced by the EJB container at run time. EJB components might be stateful, but are not by nature, contextual.

The following session bean instances are obtained using dependency injection:

- Contextual
- Bound to a life cycle context
- Available to other instances that launch in that context
- Container creates an instance when needed
- Container destroys an instance when context ends

The WebSphere Application Server CDI container performs dependency injection on all session and message-driven bean instances, even instances that are not contextual instances. WebSphere Application Server CDI supports injection of CDI beans inside enterprise beans and vice versa.

Usage

Note: Use the following best practices when injecting enterprise beans:

- Use the @Inject method for contextual injection of local session beans.
- Use the @EJB method for remote session beans

See the following examples of using the @EJB method in CDI:

Define producers making the EJB available for non-contextual injection:

```
@Produces @EJB PaymentService paymentService;
```

Consume the injected types in other CDI beans:

```
@Inject PaymentService myPaymentService
```

Practical considerations

You can define CDI-style interceptors with interceptor bindings and decorators enterprise beans. Interceptors are declared using @Interceptors methods or in ejb-jar.xml files, which are called before interceptors and are declared using interceptor bindings. Interceptors are called before decorators.

WebSphere Application Server supports failover (activation and passivation) of CDI beans and enterprise beans, along with their interceptors and decorators. EJB failover support with CDI only works for stateful session beans and requires the same configuration as stateful session bean failover. See the stateful session bean failover for the EJB container topic for more information. Configure EJB failover with web HTTP session failover. See the “Configuring for database session persistence” and the “Configuring memory-to-memory replication for the peer-to-peer mode (default memory-to-memory replication)” topics for more information. Except for abstract decorators, failover services are based on currentWebSphere Application Server failover providers. Web session failover and EJB stateful session bean failover and configured separately.

When a contextual (@Injected) instance of an EJB container is destroyed as a result of going out of scope, and if the underlying EJB container was not already removed by direct invocation of a remove method by the application, the WebSphere Application Server CDI container removes the stateful session bean.

The WebSphere Application Server CDI container removes the stateful session bean when:

- You use the @Inject method to create a contextual injection instance and that instance in an EJB container is destroyed as a result of going out of scope.
- The underlying EJB container was not already removed by direct invocation of a remove method by the application.

You must also consider the scope and state propagation of CDI beans. The request and application scope CDI beans maintain state in their respective contexts across the web and EJB containers. For instance, a request-scoped CDI bean injected in a servlet holds its state when a business method on stateful session enterprise bean accesses the same request-scoped bean.

Developing RRD extensions

Servlet extension interfaces

You can use extension generators and handlers to add content to the remote message for servlet and portlets calls. The servlet extension can also modify existing behavior by leveraging the filter concept. The Remote Request Dispatcher (RRD) extension framework relies on extension generators, which attach arbitrary data to an outbound RRD request, and extension handlers, which consume the data and perform actions based on the data received. For more information on the `com.ibm.wsspi.rrd.extension` package, refer to the Additional Application Programming Interfaces (APIs) article for administrators.

Extension generators

Extension generators, which are part of an extension generator chain, are invoked prior to initiating an RRD request. This extension generator chain is defined in the com.ibm.wsspi.rrd.generators extension point of the `plugin.xml` file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the `WEB-INF` directory of an RRD-enabled local web application.

Each extension generator is defined by a generator element, which contains an id attribute, which is used to assign a unique identifier to the extension generator such that any extension generator can be targeted by extensions added to RRD response data. The class attribute is used to specify the class name of the extension generator, which must implement the `com.ibm.wsspi.rrd.extension.generator.ExtensionGenerator` interface. Each extension generator can also have an execution order associated with it via the order attribute, which is useful for enforcing extension generator execution order in an environment where multiple extension generator descriptor files are present. Additionally, a generator has a mandatory attribute called type that defines the type of the generator. For servlet RRD, the value is “servlet” and the class must implement the `com.ibm.wsspi.rrd.extension.generator.ExtensionGenerator` interface.

Additionally, each extension generator may be provided with an arbitrary number of initialization parameters, which are specified by including zero or more `init-param` elements as children of the generator element. An example extension generator declaration follows:

```
<extension point="com.ibm.wsspi.rrd.generators">
  <generator id="int1"
        class="com.ibm.ws.rrd.example.extension.IntExtensionGenerator"
        order="1"
        type="servlet">
    <init-param>
              <param-name>intValue</param-name>
              <param-value>100</param-value>
    </init-param>
   </generator>
   <generator id="string1"
        class="com.ibm.ws.rrd.example.extension.StringExtensionGenerator"
```

```
        order="2"
        type="servlet">
      <init-param>
              <param-name>stringValue</param-name>
              <param-value>This is an example string</param-value>
      </init-param>
   </generator>
   <generator id="int2"
        class="com.ibm.ws.rrd.example.extension.IntExtensionGenerator"
        order="3"
        type="servlet">
      <init-param>
              <param-name>intValue</param-name>
              <param-value>200</param-value>
      </init-param>
   </generator>
</extension>
```

For more information on the `com.ibm.wsspi.rrd.extension.generator` package, refer to the Additional Application Programming Interfaces (APIs) article for administrators.

Extension handlers

Extension handlers, which are part of an extension handler chain, are invoked after an RRD request has been received. This extension handler chain is defined in the `com.ibm.wsspi.rrd.handlers` extension point of the `plugin.xml` file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the `WEB-INF` directory of an RRD-enabled local web application.

Each extension handler is defined by a handler element, which contains namespaceURI and localName attributes, the combination of which defines the qualified name of the extension data that the extension handler can process. Each extension handler additionally requires a unique identifier, specified by the id attribute. The value specified by this attribute must correspond to an extension generator, which generates extension data of a matching qualified name and identifier. The class attribute is used to specify the class name of the extension handler, which must implement the com.ibm.wsspi.extension.handler.ExtensionHandler interface. Additionally a handler has a mandatory attribute called type that defines the type of the handler. The value is “servlet” and the class must implement the com.ibm.wsspi.rrd.extension.handler.ExtensionHandler interface.

Additionally, each extension handler may be provided with an arbitrary number of initialization parameters, which are specified by including zero or more `init-param` elements as children of the handler element. An example extension handler declaration follows:

```
  <extension point="com.ibm.wsspi.rrd.handlers">
      <handler id="int1"
               class="com.ibm.ws.rrd.example.extension.IntExtensionHandler"
               namespaceURI="http://www.ibm.com/ws/rrd/ext/types"
               localName="SimpleType" order="1"
               type="servlet"/>
      <handler id="string1"
               class="com.ibm.ws.rrd.example.extension.StringExtensionHandler"
               namespaceURI="http://www.ibm.com/ws/rrd/ext/types"
               localName="SimpleType" order="2"
               type="servlet"/>
      <handler id="int2"
               class="com.ibm.ws.rrd.example.extension.IntExtensionHandler"
               namespaceURI="http://www.ibm.com/ws/rrd/ext/types"
               localName="SimpleType" order="3"
               type="servlet"/>
</extension>
```

For more information on the com.ibm.wsspi.rrd.extension.handler package, refer to the Additional Application Programming Interfaces (APIs) article for administrators.

Extension delegators

An extension delegator enables RRD to handle arbitrary servlet containers by allowing users to specify the specific extension generator and handler chain instances that are to be used during an RRD call. RRD maintains a user-extendable list of extension delegators and selects an appropriate delegator at runtime based on the type of servlet request being issued.

Custom extension delegators may be defined in the com.ibm.wsspi.rrd.rrd-extension-delegator extension point of the plugin.xml file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the WEB-INF directory of an RRD-enabled local web application.

Each extension delegator is defined by an ExtensionDelegator element, which contains a priority attribute for defining the relative order in which an extension delegator is initiated, and a classname attribute that defines the implementing class for a particular extension delegator, which must implement the com.ibm.wsspi.rrd.extension.factory.ExtensionDelegator interface. Note that the execution order of two or more extension delegators with the same priority is not predictable. An example extension delegator declaration follows:

```
<extension point="com.ibm.wsspi.rrd.rrd-extension-delegator">
  <ExtensionDelegatorRegistration>
    <ExtensionDelegator priority="1" classname="com.ibm.ws.rrd.extension.PortletExtensionDelegator"/>
    <ExtensionDelegator priority="2" classname="com.ibm.ws.rrd.extension.ServletExtensionDelegator"/>
  </ExtensionDelegatorRegistration>
</extension>
```

Custom EMF packages

Extension data produced by an extension generator and consumed by an extension handler is serialized using the Eclipse Modeling Framework (EMF). Users intending to use custom extension data should utilize the com.ibm.wsspi.rrd.rrd-emf-packages extension point in order to ensure that the proper EMF packages are initialized prior to use by RRD. This extension point is part of the plugin.xml file, which can reside in one of the following locations:

- Another OSGI Bundle.
- Any shared library. For example, a shared library bound to a server class loader.
- In the WEB-INF directory of an RRD-enabled local web application.

Each EMF package is defined by an emfPackage element, which contains a className element that must point to the generated EMF factory implementation class for a particular EMF package (the generated model class which implements org.eclipse.emf.ecore.impl.EFactoryImpl). An example EMF package declaration follows:

```
<extension point="com.ibm.wsspi.rrd.rrd-emf-packages">
  <emfPackages>
    <emfPackage className="com.ibm.ws.rrd.webservices.types.emf.impl.TypesFactoryImpl" />
  </emfPackages>
</extension>
```

Note: If the same generated EMF model code is shared among multiple web applications that the EMF model code must be part of a shared server library, but only in the case that all web applications are running in the same application server. In production, this is usually not the case, and common EMF model code may exist at the web application level.

Developing servlet applications using asyncrhonous request dispatcher

Developing servlet applications using asynchronous request dispatcher

Web modules can dispatch requests concurrently on separate threads. Requests can be dispatched by the server or client.

Before you begin

For additional information about the AsyncRequestDispatcherConfig and the AsyncRequestDispatcher interfaces, review the com.ibm.websphere.webcontainer.async package in the application programming interfaces (API) documentation. The generated API documentation is available in the information center table of contents from the path Reference > APIs - Application Programming Interfaces.

Review the asynchronous request dispatcher application (ARD) design considerations topic before completing the following steps.

About this task

Concurrent dispatching can improve servlet response time. If operations are dependant on each other, do not enable asynchronous request dispatching, therefore, select Disabled. Concurrent dispatching might result in errors when operations are dependant. Select Server side to enable the server to aggregate requests dispatched concurrently. Select Client side to enable the client to aggregate requests dispatched concurrently.

Procedure

1. Logically separate resource intensive operations.
2. Develop servlets that use an asynchronous request dispatcher to include these operations.
3. Enable asynchronous request dispatching on an application server.
4. Deploy the application in an application server that has asynchronous request dispatching enabled.
5. Select an aggregation type for the application that needs ARD.
6. Optional: Configure the AsyncRequestDispatcherWorkManager work manager that is used for the request dispatch threads.
7. Restart the application server.

What to do next

Restart the modified applications if already installed or start newly installed applications to enable ARD on each application.

Assembling web applications

Assembling web applications

Assemble a web module to contain servlets, JavaServer Pages (JSP) files, and related code artifacts. (Group enterprise beans, client code, and resource adapter code in separate modules). After assembling a web module, you can install it as a stand-alone application or combine it with other modules into an enterprise application.

Before you begin

This topic assumes that you have created and unit tested Servlets and JavaServer Pages (JSP) files and other web components that you want to assemble in an enterprise application and deploy onto an application server.

About this task

Use an assembly tool to assemble a web module in any of the following ways:
- Import an existing web module (WAR file).
- Create a new web module.
- Copy code artifacts (such as servlets) from one web module into a new web module.

Although you can input various properties for web archives, available properties are specific to the Servlet, JSP, and Java Platform, Enterprise Edition (Java EE) specification level.

Procedure

1. Start an assembly tool.
2. If you have not done so already, configure the assembly tool for work on Java EE modules. Ensure that **J2EE** and **Web** capabilities are enabled.
3. Migrate WAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to an assembly tool. To migrate files, import your WAR files to the assembly tool.
4. Create a new web module.
5. Copy code artifacts (such as servlets) from one web module into a new web module.

Results

A web project is migrated or created. Files for the Web project are shown in the Project Explorer view under **Enterprise Applications** and **Web Projects**.

What to do next

You can now deploy your web project to an application server.

web.xml file

The `web.xml` file provides configuration and deployment information for the web components that comprise a web application.

The Java Servlet specification defines the `web.xml` deployment descriptor file in terms of an XML schema document. For backwards compatibility, any `web.xml` file that is written to Servlet 2.2 or above that worked in previous versions of WebSphere Application Server are supported by the web container.

If you use Rational Application Developer Version 6 to create your portlets, you must remove the following reference to the `std-portlet.tld` from the `web.xml` file:

```
<taglib id="PortletTLD">
  <taglib-uri>http://java.sun.com/portlet</taglib-uri>
  <taglib-location>/WEB-INF/tld/std-portlet.tld</taglib-location>
</taglib>
```

Location

The `web.xml` file must reside in the `WEB-INF` directory under the context of the hierarchy of directories that exist for a web application.

For example, if the application is `client.war`, then the `web.xml` file is placed in the *install_root*/*client war*/`WEB-INF` directory.

Usage notes

- Is this file read-only?

  No
- Is this file updated by a product component?

  This file is updated by the assembly tool.
- If so, what triggers its update?

  The assembly tool updates the `web.xml` file when you assemble web components into a web module, or when you modify the properties of the web components or the web module.
- How and when are the contents of this file used?

  WebSphere Application Server functions use information in this file during the configuration and deployment phases of web application development.

Sample file entry

Note: The web.xml file does not represent the entire configuration that is available for the web application. There are other servlets filters, and listeners that can be defined using programmatic configurations, annotations, and web fragments.

Note: Marking the web application metadata complete will prevent annotations and web fragments from being able to configure components.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
 <display-name>Servlet 3.0 application</display-name>
 <filter>
    <filter-name>ServletMappedDoFilter_Filter</filter-name>
    <filter-class>tests.Filter.DoFilter_Filter</filter-class>
    <init-param>
      <param-name>attribute</param-name>
      <param-value>tests.Filter.DoFilter_Filter.SERVLET_MAPPED</param-value>
    </init-param>
  </filter>
 <filter-mapping>
  <filter-name>ServletMappedDoFilter_Filter</filter-name>
  <url-pattern>/DoFilterTest</url-pattern>
  <dispatcher>REQUEST</dispatcher>
 </filter-mapping>
 <filter-mapping>
  <filter-name>ServletMappedDoFilter_Filter</filter-name>
  <url-pattern>/IncludedServlet</url-pattern>
  <dispatcher>INCLUDE</dispatcher>
 </filter-mapping>
 <filter-mapping>
  <filter-name>ServletMappedDoFilter_Filter</filter-name>
  <url-pattern>ForwardedServlet</url-pattern>
  <dispatcher>FORWARD</dispatcher>
 </filter-mapping>
 <listener>
  <listener-class>tests.ContextListener</listener-class>
 </listener>
 <listener>
  <listener-class>tests.ServletRequestListener.RequestListener</listener-class>
 </listener>
 <servlet>
  <servlet-name>welcome</servlet-name>
  <servlet-class>WelcomeServlet</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>ServletErrorPage</servlet-name>
```

```
  <servlet-class>tests.Error.ServletErrorPage</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>IncludedServlet</servlet-name>
  <servlet-class>tests.Filter.IncludedServlet</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>ForwardedServlet</servlet-name>
  <servlet-class>tests.Filter.ForwardedServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>welcome</servlet-name>
  <url-pattern>/hello.welcome</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>ServletErrorPage</servlet-name>
  <url-pattern>/ServletErrorPage</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>IncludedServlet</servlet-name>
  <url-pattern>/IncludedServlet</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>ForwardedServlet</servlet-name>
  <url-pattern>/ForwardedServlet</url-pattern>
 </servlet-mapping>
 <welcome-file-list>
  <welcome-file>hello.welcome</welcome-file>
 </welcome-file-list>
 <error-page>
  <exception-type>java.lang.ArrayIndexOutOfBoundsException</exception-type>
  <location>/ServletErrorPage</location>
 </error-page>
  <error-page>
   <error-code>404</error-code>
   <location>/error404.html</location>
  </error-page>
</web-app>
```

Note: For each <error-page> declaration, select either <exception-type> or <error-code>, but not both. The <location> tag is required.

File serving

In file serving, web applications can serve static file types, such as HTML. File-serving attributes are used by the servlet that implements file-serving behavior.

The file-serving behavior is implemented by setting the fileservingenabled property to true when configuring the web module.

Example attributes:

bufferSize

Sets buffer size that is used for serving static files.

extendedDocumentRoot

Enables you to configure an application with one or more directory paths from which you can serve static files and Java ServerPages (JSP) files. You can use this attribute when an application requires access to files that exist outside of the application web application archive (WAR) directory. For example, if several applications require access to a set of common files, you can place the common files in a directory to which you can link each application as an extended document root directory.

Use this attribute in addition to the contextRoot attribute.

You can also use this attribute to define a WebSphere variable on multiple nodes to the appropriate directory.

Example:

```
<fileServingEnabled="true">
<fileServingAttributes xmi:id="FileServingAttribute_1" name="extendedDocumentRoot"
   value="${MY_CUSTOM_VARIABLE}"/>
```

where *MY_CUSTOM_VARIABLE* is the WebSphere variable that you want to define on multiple nodes.

For more information, see JSP engine configuration parameters.

file.serving.patterns.allow
Specifies that only files matching the specified pattern are served.

file.serving.patterns.deny
Specifies that files that match the specified file pattern are denied

Configuring JavaServer Faces implementation

Configuring JavaServer Faces implementation

Use this task to specify which JavaServer Faces implementation to use. You can use Apache MyFaces 2.0 or the SUN Reference Implementation 1.2 of JSF, or your own implementation.

Before you begin

Ensure that your application is configured for JavaServer Faces (JSF) using the specific `web.xml` context parameters for the implementation that you have chosen.

Attention: The JSF implementation is a server-wide configuration setting. Thus, if you have multiple applications which require different JSF implementations, you must separate the applications into different application servers or clusters, and specify the implementation by following the steps in this topic for each application server or cluster. Alternatively, you can use an isolated shared library, as described for third-party JSF implementations, as follows.

About this task

Note: The Application Server JSF engine determines if the SUN Reference Implementation (RI) 1.2 or Apache MyFaces 2.0 is used from the Application Server run time. If either is used, the correct listener class is registered with the web container. You do not need to add the com.sun.faces.ConfigureListener or the org.apache.myfaces.StartupConfigureListener to your `web.xml` file.

Be Aware: If you want to use a third-party JSF implementation that is not shipped with the product, then:

- Keep the configuration set to MyFaces.
- Add the third-party listener to the `web.xml` file that is required.
- Add the third-party implementation Java archive (JAR) files to the application as an isolated shared library and associate it with your application.

You can also configure the JSF implementation on the Provide JSP reloading options for web modules panel for application installation and update wizards.

Procedure

Configure the server or cluster to use the JSF implementation that you want. You can do this task using the administrative console or the wsadmin tool.

- In the administrative console panel, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **JSP and JSF options**

Select one of the following implementations:

- Sun Reference Implementation 1.2 - Select this option to use the Sun Reference Implementation 1.2 JSF implementation.
- MyFaces 2.0 - Select this option to use the MyFaces 2.0 JSF implementation. This option is the default JSF implementation.

- Using the wsadmin tool:
  - An example of setting a single server to use the Sun RI 1.2 JSF implementation:

```
wsadmin>set server [$AdminConfig list ApplicationServer *server1*]
server1(cells/myNode01Cell/nodes/myNode01/servers/server1|server.xml#ApplicationServer_1183122130078)
wsadmin>$AdminConfig modify $server {{jsfProvider SunRi1.2}}

wsadmin>$AdminConfig save
```

  - An example of setting a cluster to use the MyFaces 2.0 JSF implementation:

```
wsadmin>set cluster [$AdminConfig list ServerCluster]
cluster1(cells/myNode01Cell/nodes/myNode01/clusters/cluster1|cluster.xml#ServerCluster_1173916133721)
wsadmin>$AdminConfig modify $cluster {{jsfProvider MyFaces}}

wsadmin>$AdminConfig save
```

Results

What to do next

Configure JSF engine parameters as necessary.

Developing session management in servlets

About this task

This information, combined with the coding example SessionSample.java, provides a programming model for implementing sessions in your own servlets.

Procedure

1. Get the HttpSession object.

   To obtain a session, use the getSession method of the javax.servlet.http.HttpServletRequest object in the Java Servlet 3.0 API.

   When you first obtain the HttpSession object, the Session Management facility uses one of three ways to establish tracking of the session: cookies, URL rewriting, or Secure Sockets Layer (SSL) information.

   Note: Session tracking using the SSL ID is deprecated in WebSphere Application Server version 7.0. You can configure session tracking to use cookies or modify the application to use URL rewriting

   Assume the Session Management facility uses cookies. In such a case, the Session Management facility creates a unique session ID and typically sends it back to the browser as a *cookie*. Each subsequent request from this user (at the same browser) passes the cookie containing the session ID, and the Session Management facility uses this ID to find the user's existing HttpSession object.

   In Step 1 of the code sample, the Boolean(create) is set to `true` so that the HttpSession object is created if it does not already exist. (With the Servlet 2.3 API and later, the javax.servlet.http.HttpServletRequest.getSession() method with no boolean defaults to `true` and creates a session if one does not already exist for this user.)

2. Store and retrieve user-defined data in the session.

   After a session is established, you can add and retrieve user-defined data to the session. The HttpSession object has methods similar to those in java.util.Dictionary for adding, retrieving, and removing arbitrary Java objects.

In Step 2 of the code sample, the servlet reads an integer object from the HttpSession, increments it, and writes it back. You can use any name to identify values in the HttpSession object. The code sample uses the name sessiontest.counter.

Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

3. (Optional) Output an HTML response page containing data from the HttpSession object.
4. Provide feedback to the user that an action has taken place during the session. You may want to pass HTML code to the client browser indicating that an action has occurred. For example, in step 3 of the code sample, the servlet generates a web page that is returned to the user and displays the value of the sessiontest.counter each time the user visits that web page during the session.
5. (Optional) Notify Listeners. Objects stored in a session that implement the javax.servlet.http.HttpSessionBindingListener interface are notified when the session is preparing to end and become invalidated. This notice enables you to perform post-session processing, including permanently saving the data changes made during the session to a database.
6. End the session. You can end a session:
   - Automatically with the Session Management facility if a session is inactive for a specified time. The administrators provide a way to specify the amount of time after which to invalidate a session.
   - By coding the servlet to call the invalidate() method on the session object.

Example

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class SessionSample  extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {


   // Step 1: Get the Session object

      boolean create = true;
      HttpSession session = request.getSession(create);

   // Step 2: Get the session data value


      Integer ival = (Integer)
      session.getAttribute ("sessiontest.counter");
      if (ival == null) ival = new Integer (1);
      else ival = new Integer (ival.intValue () + 1);
      session.setAttribute ("sessiontest.counter", ival);

   // Step 3: Output the page

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head><title>Session Tracking Test</title></head>");
      out.println("<body>");
      out.println("<h1>Session Tracking Test</h1>");
      out.println ("You have hit this page " + ival + " times" + "<br>");
      out.println ("Your " + request.getHeader("Cookie"));
      out.println("</body></html>");
   }
}
```

Assembling so that session data can be shared

By default, the session management facility supports session scoping by web module in accordance with the Servlet 2.3 and later API specification. Only servlets in the same web module can access the data associated with a particular session. However, you can use the IBMApplicationSession object or the IBM extension, shared session context, to share data outside of the web module scope.

About this task

The IBMApplicationSession object is a parent session object that can be retrieved by a web module's session and can share session attributes across all of the web modules in a business-level application. The default scope of the business-level application is the enterprise application. The shared session context option extends the scope of the session attributes as well. Using the shared session context extension, there is only one session object for the entire business-level application or for the default enterprise application.

If you are using a shared session for a business-level application, then the class files for all objects placed in the session must exist in an isolated shared library and be common among all applications.

The benefit to using the IBMApplicationSession method is that each web module can maintain its own session as well as have a reference to the shared session.

If you're migrating an application from a previous version of the product, the IBMApplicationSession method requires a change to the application logic of the application.

Restriction: To use a shared session, you must install all applications within a business-level application on a given server. You cannot split up the enterprise application by servers. For example, you cannot use this option when one enterprise application in "BLA1" is installed on one server and a second enterprise application also in "BLA1" is installed on a different server. In such split installations, applications might share session attributes across web modules using distributed sessions, but session data integrity is lost when concurrent access to a session is made in different web modules. It also severely restricts use of some session management features, like TIME_BASED_WRITES.

For enterprise applications on which this shared session context extension is enabled, the session management configuration on the web module inside the enterprise application is ignored. Then session management configuration defined on enterprise application is used if session management is overwritten at the enterprise application level. Otherwise, the session management configuration on the web container is used. If using multiple enterprise applications within a business-level application, the session management configuration must be common among all applications and web modules within this business-level application.

HttpSession listeners that are defined in all the web modules inside the business-level application or enterprise application are invoked for session events. The order of listener invocation is not guaranteed.

Complete the following to share session data across the business-level application.

Procedure

1. Complete the following to share session data using the IBMApplicationSession object within the application code.
   a. Retrieve the session object
      ```
      HttpSession session = request.getSession();
      ```
   b. Cast this object to an IBMSession object and call the getIBMApplicationSession method.
      ```
      IBMApplicationSession appSession = ((IBMSession)session).getIBMApplicationSession();
      ```
   c. Use the appSession like a normal session object.

2. Do the following to share session data using the Shared session context extension.
   a. Start an assembly tool.
   b. In the assembly tool, right-click the application (EAR file) that you want to share and click **Open With > Deployment Descriptor Editor**.
   c. In the application deployment descriptor editor of the assembly tool, select **Shared session context** under WebSphere Extensions. Make sure the class definition of attributes put into session are available to all web modules in the enterprise application. The shared session context does not fully meet the requirements of the specifications.
   d. Save the application (EAR) file. In the assembly tool, after you close the application deployment descriptor editor, confirm that you want to save changes made to the application.

Chapter 28. Developing web services

This page provides a starting point for finding information about web services.

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. They implement a services oriented architecture (SOA), which supports the connecting or sharing of resources and data in a very flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

Using JAXB for XML data binding

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified web services development. JAXB provides the xjc schema compiler, the schemagen schema generator and a runtime framework to support marshalling and unmarshalling of XML documents to and from Java objects.

About this task

JAXB is an XML-to-Java binding technology that enables transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB technology consists of a runtime API and accompanying tools that simplify access to XML documents. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides tooling to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the XML data structure.

JAXB is the default data binding technology used by the Java API for XML Web Services (JAX-WS) tooling and implementation within this product. You can develop JAXB objects to use within your JAX-WS applications. You can also use JAXB independently of the JAX-WS programming model as a convenient way to leverage the XML data binding technology to manipulate XML within your Java applications.

JAXB is also the default data binding technology used by Service Component Architecture (SCA) applications. JAXB enables the SCA service implementation side and the SCA client reference side to interact with Java objects without worrying about how the data is transformed into and from XML.

Note: This version of the application server supports the JAXB 2.2 specification. JAX-WS 2.2 requires JAXB 2.2 for data binding. JAXB 2.2 provides minor enhancements to its annotations for improved schema generation and better integration with JAX-WS.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

JAXB provides the **xjc** schema compiler tool, the **schemagen** schema generator tool, and a runtime framework. The **xjc** schema compiler tool enables you to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. You can also start with a set of JavaBeans and use the **schemagen** schema generator tool to create the XML schema. After using either the schema compiler or the schema generator command-line tools, you can convert your XML documents both to and from Java objects and use the resulting Java classes to assemble a web services application.

In addition to using the tools from the command-line, you can invoke these JAXB tools from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the **xjc** schema compiler tool. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the

Ant build environment to invoke the **schemagen** schema generator tool. These Ant tasks require that the com.ibm.jaxb.tools.jar and the com.ibm.jaxws.tools.jar files be in the classpath.

JAXB annotated classes and artifacts contain all the information that the JAXB runtime API needs to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML files and unmarshaling the XML document back to JAXB class instances. The JAXB binding package, `javax.xml.bind`, defines the abstract classes and interfaces that are used directly with content classes. In addition the package defines the marshal and unmarshal APIs.

You can optionally use JAXB binding customizations to override the default generated type mappings. You can customize JAXB bindings using inline annotations in the source schema, or by using an external bindings customization file to pass your customizations to the JAXB binding compiler, xjc, to control the Java type mappings. Alternatively, you can add Java annotations to existing Java classes to pass to the schema generator, schemagen, to control the schema or XML type mappings. See the JAXB specification for information regarding binding customization options and Java annotations.

Using JAXB, you can manipulate data objects in the following ways:

Procedure

- Generate an XML schema from a Java class. Use the schema generator **schemagen** command to generate an XML schema from Java classes.
- Generate Java classes from an XML schema. Use the schema compiler **xjc** command to create a set of JAXB-annotated Java classes from an XML schema.
- Marshal and unmarshal XML documents. After the mapping between XML schema and Java classes exists, use the JAXB binding runtime to convert XML instance documents to and from Java objects.

Results

You now have JAXB objects that your Java application can use to manipulate XML data.

Using JAXB schemagen tooling to generate an XML schema file from a Java class

Use Java Architecture for XML Binding (JAXB) schemagen tooling to generate an XML schema file from Java classes.

Before you begin

Identify the Java classes or a set of Java objects to map to an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between Java classes and XML schema. XML schema documents describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop web services using a bottom-up development approach starting from existing JavaBeans or enterprise beans, use the **wsgen** tool to generate the artifacts for Java API for XML-Based Web Services (JAX-WS) applications. After the Java artifacts for your application are generated, you can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, **schemagen** command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The

XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

You can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, **schemagen** command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

Note: This product supports the JAXB 2.2 specification. JAX-WS 2.2 requires JAXB 2.2 for data binding.

JAXB provides compilation support to enable you to configure the **schemagen** schema generator so that it does not automatically generate a new schema. This is helpful if you are using a common schema such as the World Wide Web Consortium (W3C), XML Schema, Web Services Description Language (WSDL), or WS-Addressing and you do not want a new schema generated for a particular package that is referenced. The `location` attribute on the @XmlSchema annotation causes the schemagen generator to refer to the URI of the existing schema instead of generating a new one.

In addition to using the **schemagen** tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the **schemagen** schema generator tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Note: When running the **schemagen** tool, the schema generator does not correctly read the @XmlSchema annotations from the package-info class file to derive targetNamespaces. Instead of using the @XMLSchema annotation, use one of the following methods:

- Provide a package-info.java file with the @XmlSchema; for example:

```
schemagen sample.Address sample\package-info.java
```

- Use the @XmlType annotation namespace attribute to specify a namespace; for example:

```
@XmlType(namespace="http://myNameSpace")
```

Procedure

1. Locate the Java source files or Java class files to use in generating an XML schema file. Ensure that all classes referenced by your Java class files are contained in the classpath or are provided to the tool using the`-classpath/-cp` options.
2. Use the JAXB schema generator, **schemagen** command to generate an XML schema. The schema generator is located in the *app_server_root*\bin\ directory.

The parameters, `myObj1.java` and `myObj2.java`, are the names of the Java files containing the data objects. If `myObj1.java` or `myObj2.java` refer to Java classes that are not passed into the **schemagen** command, you must use the `-cp` option to provide the classpath location for these Java classes.Read about the **schemagen** command to learn more about this command and additional options that you can specify.

3. (Optional) Use JAXB program annotations defined in the `javax.xml.bind.annotations` package to customize the JAXB XML schema mappings.
4. (Optional) Configure the `location` property on the @XmlSchema annotation to indicate to the schema compiler to use an existing schema rather than generating a new one. For example,

```
@XmlSchema(namespace="foo")
package foo;
@XmlType
class Foo {
@XmlElement Bar zot;
}
@XmlSchema(namespace="bar",
location="http://example.org/test.xsd")
package bar;
@XmlType
class Bar {
...
}
<xs:schema targetNamespace="foo">
<xs:import namespace="bar"
schemaLocation="http://example.org/test.xsd"/>
<xs:complexType name="foo">
<xs:sequence>
<xs:element name="zot" type="bar:Bar" xmlns:bar="bar"/>
</xs:sequence>
</xs:complex
```

the `location="http://example.org/test.xsd"` indicates the location on the existing schema to the **schemagen** tool and a new schema is not generated.

Results

Now that you have generated an XML schema file from Java classes, you are ready to marshal and unmarshal the Java objects as XML instance documents.

Note: The schemagen command does not differentiate the XML namespace between multiple @XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

Example

The following example illustrates how JAXB tooling can generate an XML schema file from an existing Java class, Bookdata.java.

1. Copy the following `Bookdata.java` file to a temporary directory.

```
package generated;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.datatype.XMLGregorianCalendar;



@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "bookdata", propOrder = {
    "author",
```

```
    "title",
    "genre",
    "price",
    "publishDate",
    "description"
})
public class Bookdata {

    @XmlElement(required = true)
    protected String author;
    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String genre;
    protected float price;
    @XmlElement(name = "publish_date", required = true)
    protected XMLGregorianCalendar publishDate;
    @XmlElement(required = true)
    protected String description;
    @XmlAttribute
    protected String id;

     public String getAuthor() {
        return author;
    }
    public void setAuthor(String value) {
        this.author = value;
    }
    public String getTitle() {
        return title;
    }

     public void setTitle(String value) {
        this.title = value;
    }


    public String getGenre() {
        return genre;
    }

    public void setGenre(String value) {
        this.genre = value;
    }


    public float getPrice() {
        return price;
    }


    public void setPrice(float value) {
        this.price = value;
    }


    public XMLGregorianCalendar getPublishDate() {
        return publishDate;
    }


    public void setPublishDate(XMLGregorianCalendar value) {
        this.publishDate = value;
    }


    public String getDescription() {
        return description;
    }


    public void setDescription(String value) {
        this.description = value;
    }


    public String getId() {
        return id;
    }


    public void setId(String value) {
        this.id = value;
    }

}
```

2. Open a command prompt.

3. Run the schemagen schema generator tool from the directory where you copied the `Bookdata.java` file.
4. The XML schema file, `schema1.xsd` is generated:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="bookdata">
    <xs:sequence>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="genre" type="xs:string"/>
      <xs:element name="price" type="xs:float"/>
      <xs:element name="publish_date" type="xs:anySimpleType"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Refer to the JAXB Reference implementation documentation for additional information about the **schemagen** command.

Using JAXB xjc tooling to generate JAXB classes from an XML schema file

Use Java Architecture for XML Binding (JAXB) xjc tooling to compile an XML schema file into fully annotated Java classes.

Before you begin

Develop or obtain an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. XML schemas describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop web services using a top-down development approach starting with an existing Web Services Description Language (WSDL) file, use the **wsimport** tool to generate the artifacts for your Java API for XML-Based Web Services (JAX-WS) applications when starting with a WSDL file. After the Java artifacts for your application are generated, you can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, **xjc** command-line tool. The resulting annotated Java classes contain all the necessary information that the JAXB runtime requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web Services (JAX-WS) applications or other Java applications for processing XML data.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

Note: This product supports the JAXB 2.2 specification. JAX-WS 2.2 requires JAXB 2.2 for data binding.

In addition to using the **xjc** tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the **xjc** schema compiler tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Note: If you are using the **xjc** Ant task, you must use the `destdir` parameter to specify the destination directory instead of the `target` option. Specifying the `target` option when using the **xjc** Ant task causes an error.

Procedure

1. Use the JAXB schema compiler, **xjc** command to generate JAXB-annotated Java classes. The schema compiler is located in the *app_server_root*\bin\ directory. The schema compiler produces a set of packages containing Java source files and JAXB property files depending on the binding options used for compilation.
2. (Optional) Use custom binding declarations to change the default JAXB mappings. Define binding declarations either in the XML schema file or in a separate bindings file. You can pass custom binding files by using the **-b** option with the **xjc** command.
3. Compile the generated JAXB objects. To compile generated artifacts, add the Thin Client for JAX-WS with WebSphere Application Server to the classpath.

Results

Now that you have generated JAXB objects, you can write Java applications using the generated JAXB objects and manipulate the XML content through the generated JAXB classes.

Example

The following example illustrates how JAXB tooling can generate Java classes when starting with an existing XML schema file.

1. Copy the following `bookSchema.xsd` schema file to a temporary directory.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="CatalogData">
     <xsd:complexType >
       <xsd:sequence>
         <xsd:element name="books" type="bookdata" minOccurs="0"
    maxOccurs="unbounded"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
   <xsd:complexType name="bookdata">
     <xsd:sequence>
       <xsd:element name="author" type="xsd:string"/>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="genre" type="xsd:string"/>
       <xsd:element name="price" type="xsd:float"/>
       <xsd:element name="publish_date" type="xsd:dateTime"/>
       <xsd:element name="description" type="xsd:string"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:string"/>
   </xsd:complexType>
 </xsd:schema>
```

2. Open a command prompt.
3. Run the JAXB schema compiler, **xjc** command from the directory where the schema file is located. The **xjc** schema compiler tool is located in the *app_server_root*\bin\ directory.

   Running the **xjc** command generates the following JAXB Java files:

```
generated\Bookdata.java
generated\CatalogdData.java
generated\ObjectFactory.java
```

4. Use the generated JAXB objects within a Java application to manipulate XML content through the generated JAXB classes.

Refer to the JAXB Reference implementation documentation for additional information about the **xjc** command.

Using the JAXB runtime to marshal and unmarshal XML documents

Use the Java Architecture for XML Binding (JAXB) run time to manipulate XML instance documents.

Before you begin

Use JAXB to generate Java classes from an XML schema with the schema compiler, **xjc** command or to generate an XML schema from a Java class with the schema generator, **schemagen** command.

About this task

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. After data bindings exist, use the JAXB binding runtime API to convert XML instance documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the data structure. JAXB annotated classes and artifacts contains all the information that the JAXB runtime API needs to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML and unmarshaling the XML document back to JAXB class instances.

Procedure

- Marshal JAXB objects to XML instance documents.

  Use the JAXB runtime API to marshal or convert JAXB object instances into an XML instance document.

  1. Instantiate your JAXB classes.
  2. Invoke the JAXB marshaller.

  This example demonstrates how to instantiate the generated JAXB objects within an application and use the JAXBContext class and the JAXB runtime marshaller APIs to marshal the JAXB objects into XML instances.

```
JAXBContext jc = JAXBContext.newInstance("myPackageName");
//Create marshaller
Marshaller m = jc.createMarshaller();
//Marshal object into file.
m.marshal(myJAXBObject, myOutputStream);
```

  The JAXB Reference Implementation introduces additional vendor specific marshaller properties such as namespace prefix mapping, indentation, and character escaping control that are not defined by the JAXB specification. Use these properties to specify additional controls of the marshaling process. These properties operate with the JAXB Reference Implementation only and might not with other JAXB providers. Additional information regarding the vendor specific properties is located in the Java Architecture for XML Binding JAXB RI Vendor Extensions Runtime Properties specification.

- Unmarshal XML files to JAXB objects.

  Use the JAXB runtime API to unmarshal or convert an XML instance document to JAXB object instances.

  1. Obtain an existing XML instance document.
  2. Invoke the JAXB unmarshaller.

  This example demonstrates a program that reads an XML document and unmarshals or converts the XML document into JAXB object instances. Use the JAXBContext class and JAXB runtime Unmarshaller APIs to unmarshal the XML document.

```
JAXBContext jc = JAXBContext.newInstance("myPackageName");
//Create unmarshaller
Unmarshaller um = jc.createUnmarshaller();
//Unmarshal XML contents of the file myDoc.xml into your Java
  object instance.
MyJAXBObject myJAXBObject = (MyJAXBObject)
um.unmarshal(new java.io.FileInputStream( "myDoc.xml" ));
```

Results

You can now marshal JAXB Java classes, and unmarshal XML data using the JAXB binding framework. Refer to the JAXB Reference implementation documentation for additional information about the marshal and unmarshal runtime APIs

Note: If Java 2 Security is enabled, wrap your JAXBContext.newInstance(), Unmarshaller.unmarshal() and, Marshaller.marshal() method calls within a AccessController.doPrivileged method to avoid a security exception.

.

xjc command for JAXB applications

Use the Java Architecture for XML Binding (JAXB) tools to generate Java classes from an XML schema with the xjc schema compiler tool.

JAXB is an XML-to-Java binding technology that enables transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB technology consists of a runtime API and accompanying tools that simplify access to XML documents. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides a runtime environment to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the XML data structure.

You can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, **xjc** command-line tool. Use the **xjc** schema compiler tool to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. Once the mapping between XML schema and Java classes exists, XML instance documents can be converted to and from Java objects through the use of the JAXB binding runtime API. The resulting annotated Java classes contains all the necessary information that the JAXB runtime requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web Services (JAX-WS) applications or in your non-JAX-WS Java applications for processing XML data.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

In addition to using the **xjc** tool from the command line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the **xjc** schema compiler tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Note: If you are using the **xjc** Ant task, you must use the `destdir` parameter to specify the destination directory instead of the `target` option. Specifying the `target` option when using the **xjc** Ant task causes an error.

Syntax

The command-line syntax is:

If a directory is specified, all schema files in the directory are compiled.

Parameters

The `schema file/URL` JAR file name or location of the directory is the only parameter that is required. The following parameters are optional for the **xjc** command:

-b <file_name or directory>
Specifies the external JAX-WS or JAXB binding files. You can specify multiple JAX-WS and JAXB binding files by using the `-b` option; however, each file must be specified with its own `-b` option. If a directory is specified, `**/*.xjb` is searched.

-catalog <file_name>
Specifies the catalog file to resolve external entity references. It supports the TR9401, XCatalog, and the OASIS XML Catalog formats.

-classpath <path>
Specifies the location of the class files.

-d <directory>
Specifies where to place the generated output files.

-dtd
Specifies to treat the input as XML Document Type Definition (DTD). This option is unsupported and experimental.

-extension
Specifies whether to enable custom extensions for functionality not specified by the JAXB specification. Use of the extensions can result in applications that are not portable or do not interoperate with other implementations.

-help
Displays the help menu.

-httpproxy <[user[:password]@]<proxyhost>:<proxyport>>
Specifies an HTTP or HTTPs proxy.

-httpproxyfile <file_name>
This parameter is similar to the `-httpproxy` parameter, but takes the argument in a file to protect the password.

-no-header
Specifies to suppress the generation of a file header with a timestamp.

-npa
Specifies to suppress the generation of the `**/package-info.java` package level annotation.

-nv
Specifies to not perform a strict validation of the input schemas.

-p <package_name>
Specifies a target package.

-quiet
Specifies to suppress the output from the **xjc** tool.

-relaxng
Specifies to treat the input as REgular LAnguage for XML Next Generation (RELAX NG). This option is unsupported and experimental.

-readOnly
Specifies that the generated files are in read-only mode.

-relaxng-compact
Specifies to treat the input as REgular LAnguage for XML Next Generation (RELAX NG) compact syntax. This option is unsupported and experimental.

-target <version>
Specifies to generate output to conform to the specified level of the JAX-WS specification. Specify `2.0` or `2.1` for the tool to generate compliant code for the JAXB 2.0 or JAX-WS 2.1 specification respectively. Specify `2.1` for the tool to generate compliant code for the JAXB 2.1 specification. The default target version is `2.2` and generates compliant code for the JAXB 2.2 specification.

-verbose
Specifies to output messages about what the compiler is doing.

-version
Prints the version information. If you specify this option, only the version information is output and typical command processing does not occur.

-wsdl
Specifies to treat the input as a Web Services Description Language (WSDL) file and compile schemas inside the WSDL. This option is unsupported and experimental.

-xmlschema
Specifies to treat the input as a World Wide Web Consortium (W3C) XML schema. This value is the default.

schemagen command for JAXB applications

Use the schema generator tool, schemagen, to generate an XML schema using Java Architecture for XML Binding (JAXB).

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. XML schemas describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

You can generate a schema file from Java classes using the **schemagen** schema generator tool to create the XML schema. After the mapping between XML schema and Java classes exists, XML instance documents can be converted to and from Java objects through the use of the JAXB binding runtime API. The resulting Java classes contain all the necessary information that the JAXB run time requires to parse the XML for marshaling and unmarshaling. You can use the JAXB classes within Java API for XML Web Services (JAX-WS) applications or in your non-JAX-WS Java applications for processing XML data.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other

compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

Note: When running the **schemagen** tool to process JavaBeans, the schema generator will not automatically process the @XmlSchema annotations from existing package-info.class files to derive targetNamespaces. To assure that the **schemagen** tool processes namespace values correctly, use one of the following methods:

- Explicitly specify the package-info.java source file in the **schemagen** invocation; for example:

```
schemagen sample.Address sample\package-info.java
```

- Use the @XmlType annotation namespace attribute within your JavaBeans to specify a namespace; for example:

```
@XmlType(namespace="http://myNameSpace")
public class Address {...}
```

In addition to using the **schemagen** tool from the command line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the **schemagen** schema generator tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Syntax

The command-line syntax is:

Parameters

The following parameters are optional for the **schemagen** command:

-classpath <path>
Specifies the location of the Java source or class files.

-cp <path>
Specifies the location of the Java source or class files.

-d <path>
Specifies where to place the processor and the generated Java class files.

-episode<file_name>
Specifies to generate an episode file for separate compilation.

-encoding <encoding>
Specifies to use encoding when invoking the -apt or -javac tool. This property is applicable for JAXB 2.2 and later.

-help
Displays the help menu.

-version
Prints the version information. If you specify this option, only the version information is output and typical command processing does not occur.

Developing JAX-WS web services (bottom-up)

Setting up a development environment for web services

The application server provides command-line tools to develop web services clients and implementations that are based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. You must set up your development environment before you start developing web services.

Before you begin

Before you can set up a web services development environment within WebSphere Application Server, you must install WebSphere Application Server. For detailed information on installing the application server, read about installing your application server environment.

About this task

Set up a web services development environment by completing the following actions.

Procedure

1. Set up the environment.
   Run the **setupCmdLine** script from the */profile_root/<application_server>/bin* directory.
2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:

```
set PATH=%WAS_PATH%;%PATH%
```

Results

You have set up an environment so that you can develop Web services.

What to do next

Implement web services applications. See the task overview for implementing web services applications information to learn about how to develop and implement a Java EE web service.

Developing JAX-WS web services with annotations

Java API for XML-Based Web Services (JAX-WS) supports two different service endpoint implementations types, the standard web service endpoint interface and a new Provider interface to enable services to work at the XML message level. By using annotations on the service endpoint implementation or client, you can define the service endpoint as a web service.

Before you begin

Set up a development environment for web services.

About this task

This task is a required step to develop JAX-WS web services.

JAX-WS technology supports the implementation of web services based on both the standard service endpoint interface and a new Provider interface. JAX-WS endpoints are like the endpoint implementations in the Java API for XML-based RPC (JAX-RPC) specification. Unlike JAX-RPC, the requirement for a service endpoint interface (SEI) is optional for JAX-WS web services. JAX-WS services that do not have an associated SEI are regarded as having an implicit SEI; whereas services that have an associated SEI are regarded as having an explicit SEI. The service endpoint interfaces required by JAX-WS are also more

generic than the service endpoint interfaces required by JAX-RPC. With JAX-WS, the SEI is not required to extend the `java.rmi.Remote` interface as required by the JAX-RPC specification.

The JAX-WS programming model also uses support for annotating Java classes with metadata to define a service endpoint implementation as a web service and define how a client can access the web service. JAX-WS supports annotations based on the Metadata Facility for the Java Programming Language (JSR 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.0 (JSR 224) specification, which includes Java Architecture for XML Binding (JAXB) annotations. Using annotations, the service endpoint implementation can independently describe the web service without requiring a WSDL file. Annotations can provide all the WSDL information necessary to configure your service endpoint implementation or web services client. You can specify annotations on the service endpoint interface used by the client and the server, or on the server-side service implementation class.

For details regarding the supported standards and specifications, see the web services specifications and API documentation.

When developing a JAX-WS web service starting from existing Java classes, known as the bottom-up approach, you must annotate the class with either the `@WebService` (`javax.jws.WebService`) annotation or `@WebServiceProvider` (`javax.xml.ws.Provider`) annotation to initially define the class as a web service. The `@WebService` annotation defines the service as an SEI-based endpoint, while the `@WebServiceProvider` annotation defines the service as a Provider-based endpoint.

Develop SEI-based JAX-WS web services

For an SEI-based endpoint, the service endpoint interface (SEI), whether it is a Java class or a Java interface, declares the business methods provided by a particular web service. The only methods that a web services client can invoke on a JAX-WS endpoint are the business methods that are defined in the explicit or implicit SEI.

All SEI-based endpoints are required to have the `@WebService` annotation included on the implementation class. If the service implementation uses an explicit SEI, then that interface must be referenced by the `endpointInterface` attribute on the `@WebService` annotation. If the service implementation does not use an explicit SEI, then the service is described implicitly by the implementation class and is an implicit SEI.

Develop JAX-WS web services using the Provider interface

The JAX-WS programming model introduces the Provider interface for Provider endpoints, `javax.xml.ws.Provider`, as the dynamic alternative to SEI-based endpoints. The Provider interface supports a more messaging-oriented approach to web services. With the Provider interface, you can create a Java class that implements a simple interface to produce a generic service implementation class. The Provider interface defines one method, the `invoke` method, which uses generics to control both the input and output types when working with various messages or message payloads. All Provider endpoints must be annotated with the `@WebServiceProvider` (`javax.xml.ws.WebServiceProvider`) annotation. A service implementation cannot specify the `@WebService` annotation if it implements the `javax.xml.ws.Provider` interface.

Starting with WebSphere Application Server Version 7.0 and later, Java EE 5 application modules (web application modules version 2.5 or above, or EJB modules version 3.0 or above) are scanned for annotations to identify JAX-WS services and clients. However, pre-Java EE 5 application modules (web application modules version 2.4 or before, or EJB modules version 2.1 or before) are not scanned for JAX-WS annotations, by default, for performance considerations. In the Version 6.1 Feature Pack for Web Services, the default behavior is to scan pre-Java EE 5 web application modules to identify JAX-WS services and to scan pre-Java EE 5 web application modules and EJB modules for service clients during application installation. Because the default behavior for WebSphere Application Server Version 7.0 and later is to not scan pre-Java EE 5 modules for annotations during application installation or server startup, to preserve backward compatability with the feature pack from previous releases, you must configure

either the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a web application archive (WAR) file or EJB module or define the Java virtual machine custom property, com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations information.

Procedure

1. Determine if you want to define your web services using SEI endpoints or the Provider interface. If you prefer a high-level Java-centric abstraction that hide the details of converting between Java objects and their XML representation, consider using SEI-based endpoints to develop your web services. However, if you prefer that your web service operate more at the XML message level, consider using Provider-based endpoints.
2. Annotate the service endpoints.
   a. Annotate the SEI-based endpoint with the `javax.jws.WebService` annotation.
      - For an SEI-based endpoint, annotate the implementation class with the `javax.jws.WebService` annotation. You can choose to explicitly reference a service endpoint interface by defining the `@WebService.endpointInterface` attribute, which specifies this endpoint is an explicit SEI. In this case, the Java interface that is referenced must also contain the `javax.jws.WebService` annotation. If the `endpointInterface` attribute is not defined or is empty, the implementation bean is considered an implicit SEI. You can add the `@WebMethod` annotation to methods of a service endpoint interface to customize the Java-to-WSDL mappings. All public methods are considered exposed methods regardless of whether the `@WebMethod` annotation is specified. It is incorrect to have an `@WebMethod` annotation on a service endpoint interface that contains the `exclude` attribute.
      - If you use an implicit SEI, you can apply more granular control on how the methods are exposed through selective use of the `@WebMethod` annotation. For a more detailed explanation, see the exposing methods in SEI-based JAX-WS web services information.
   b. Annotate the Provider-based endpoint with the `javax.xml.ws.WebServiceProvider` annotation.
      - For a Provider-based endpoint, annotate the implementation class with the `javax.xml.ws.WebServiceProvider` annotation. This annotation must be specified only on a class that implements a strongly typed `javax.xml.ws.Provider.interface` class such as `Provider<Source>` or `Provider<SOAPMessage>`, in contrast to a class that is unbounded, such as `Provider<T>`. A strongly typed class is one that is associated with a specific input and output Java type, such as `Source` or `SOAPMessage`, for example.

```
@WebServiceProvider(
        serviceName="StringProviderService",
        wsdlLocation="META-INF/echostring.wsdl",
        targetNamespace="http://stringprovider.sample.test.org")
```

      - (optional) According the JAX-WS 2.2 specification, if you are defining a Provider-based endpoint so that the Provider implementation returns a null value, no response is needed. If the `javax.xml.ws.WebServiceProvider` annotation does not specify a WSDL, and the Provider `invoke()` method returns a null value, the default behavior of the JAX-WS runtime environment is to return a response that consists of a `SOAPEnvelope` that contains an empty `SOAPBody`. You can set the JVM property, `jaxws.provider.interpretNullAsOneway`, to `true` if you want the JAX-WS runtime environment to interpret this scenario as a request-only operation and not return a response.
3. Understand and apply best practices for exposing methods as operations in SEI-based JAX-WS web services.

   Because of ambiguity across multiple web services specifications regarding which methods are exposed as web services operations for SEI-based endpoints, you can ensure consistent behavior by following best practices, regardless of the JAX-WS implementation that you use.

Results

You have defined the service endpoint implementation that represents the web services application. See the JAX-WS annotations documentation to learn more about the supported JAX-WS annotations.

Sample JavaBeans service endpoint implementation and interface

The following example illustrates a simple explicit JavaBeans service endpoint implementation and the associated service endpoint interface:

```
/** This is an excerpt from the service implementation file, EchoServicePortTypeImpl.java.
package com.ibm.was.wssample.echo;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;

@javax.jws.WebService(serviceName = "EchoService", endpointInterface =
"com.ibm.was.wssample.echo.EchoServicePortType", targetNamespace="http://com/ibm/was/wssample/echo/",
portName="EchoServicePort")
public class EchoServicePortTypeImpl implements EchoServicePortType {

                public EchoServicePortTypeImpl() {
                }

                public String echo(String obj) {
                        String str;
                        ....
                        str = obj;
                        ....

                        return str;

                }

}
/**  This is a sample EchoServicePortType.java service interface. */

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.*;


@WebService(name = "EchoServicePortType", targetNamespace = "http://com/ibm/was/wssample/echo/",
wsdlLocation="WEB-INF/wsdl/Echo.wsdl")
public interface EchoServicePortType {


    /** ...the method process ...*/
        @WebMethod

}
```

The following example illustrates a simple Provider service endpoint interface for a Java class:

```
package jaxws.provider.source;

import javax.xml.ws.Provider;
import javax.xml.ws.WebServiceProvider;
import javax.xml.transform.Source;

@WebServiceProvider()
public class SourceProvider implements Provider<Source> {

    public Source echo(Source data) {
        return data;
    }
}
```

In the Provider implementation example, the `javax.xml.transform.Source` type is specified in the generic `<Source>` method. The generic `<Source>` method specifies that both the input and output types are `Source` objects.

What to do next

Develop Java artifacts for JAX-WS applications from JavaBeans. To learn more, see the generating Java artifacts for JAX-WS applications information.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*`/AppServer` and *configuration_root*`/DeploymentManager`.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is `/wasv8config/`*cell_name*`/`*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as *app_server_root*`/profiles/`*profile_name*. On z/OS, this will always be *app_server_root*`/profiles/default` because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

  The corresponding product variable is *smpe.install.root*.

  The default is `/usr/lpp/zWebSphere/V8R5`.

Exposing methods in SEI-based JAX-WS web services

You can use the @WebService and @WebMethod annotations on a service endpoint implementation to specify Java methods that you want to expose as Java API for XML-Based Web Services (JAX-WS) web services.

Before you begin

JAX-WS technology enables the implementation of web services based on both the standard service endpoint interface and a Provider interface. When developing a JAX-WS web service starting from existing

Java classes, known as the bottom-up approach, you must annotate the class with either the @WebService or @WebServiceProvider annotation to initially define the class as a web service.

Using the Provider interface is the dynamic approach to defining your JAX-WS services. To use the Provider interface, your class must implement the javax.xml.ws.Provider interface, and contain the @WebServiceProvider annotation. The Provider interface has one method, the invoke method, which uses generics in the Java programming language to control both the input and output types when working with various messages or message payloads.

In contrast, this topic describes how you can use Java annotations to describe your web services using the service endpoint interface (SEI) approach.

About this task

To initially define a web service, annotate the Java class with the @WebService annotation. You can also selectively annotate individual methods with the @WebMethod annotation to control their exposure as web services operations.

Because of ambiguities across multiple web services specifications regarding how methods are exposed as operations, use the following guidelines to help ensure consistent behavior regardless of the JAX-WS implementation that you use.

- To define a basic web service, annotate the implementation class with the @WebService annotation.
- To define your web services using an explicit SEI, explicitly reference a Java interface class using the endpointInterface attribute of the @WebService annotation.
- Provide a reference to a WSDL file in the wsdlLocation attribute of the @WebService annotation. By specifying a pre-defined WSDL file, performance is improved. Also, discrepancies between the WSDL file and the annotations are reported to you by the runtime environment.
- When you use an explicit SEI, all public methods in the SEI and inherited classes are always exposed. You only need to add @WebMethod annotations if you want to further customize the methods that are already exposed.
- Providing a reference in the @WebService annotation to an explicit SEI or to an existing WSDL file helps to remove possible ambiguities when exposing methods.
- If you do not use an explicit SEI, follow these rules to ensure that your methods are exposed consistently:
  - Add an @WebService annotation to your implementation class and all its superclasses that contain methods that you want to expose. Adding an @WebService annotation to a class exposes all public methods in that class that are not static or final.
  - If you want more granular control to expose only certain methods, use the @WebMethod annotation on selected methods. To ensure that a method is exposed, annotate it with the @WebMethod annotation. If you want to make sure that a method is not exposed, annotate it with the @WebMethod(exclude=true) annotation.

Note:

Behavior change for exposing methods that are not annotated:

The behavior of JAX-WS has changed regarding exposing methods as web services operations. This complies with recent clarifications to JAX-WS specifications.

Applications without an explicit SEI or WSDL that are migrated from prior versions might have additional operations exposed as shown below. You can set a property so the JAX-WS runtime environment uses the legacy behavior. You might need this when migrating applications without a WSDL or an SEI so that additional methods are not exposed.

```
@WebService
public class Foo {
 @WebMethod
 public void a() {} // exposed now, exposed before
 public void b() {} // exposed now, not exposed before
}
```

Using the new interpretation, public methods in an implementation class and its superclasses are only exposed under the following conditions:

- The containing class has an @WebService annotation.
- The method does not have an @WebMethod(exclude=true) annotation.

Using the legacy interpretation, a method in an implementation class and its superclasses are only exposed under the following conditions:

- The containing class has an @WebService annotation.
- The method has no @WebMethod annotations AND no other methods have @WebMethod annotations.
- The method has an @WebMethod or @WebMethod(exclude=false) annotation.

To specify that the JAX-WS runtime environment use the legacy @WebMethod behavior, configure the `jaxws.runtime.legacyWebMethod=true` property. You can configure this property as a Java Virtual Machine (JVM) system property or as a property in the `META-INF/MANIFEST.MF` file of a web application archive (WAR) file. By default, this property is set to `false` and the application server uses the new behavior.

You might encounter a WSWS7054E error message if all of the following conditions are true:

- Your web service application consists of unannotated methods.
- The methods are not meant to be mapped to a web service operation.
- Your application does not reference an SEI nor package a WSDL file.

The error message contains information that is similar to the following text:

```
javax.xml.ws.WebServiceException: WSWS7054E:
 The Web Services Description Language (WSDL) file could not be generated for the XXXX Web service implementation
 class because of the following error: javax.xml.ws.WebServiceException: Unable to create JAXBContext
```

The JAX-WS tooling complies with the JAX-WS specification with respect to @WebMethod mapping principles. This change might affect applications that have been dependent on previously non-compliant default behavior. If your applications package and reference WSDL or an SEI and have ALL methods correctly annotated with the @WebMethod exclude flag in the SEI implementation, then this change does not affect you. However, if you are affected, add explicit annotations to your methods to ensure that they are excluded in WSDL generation. For example: @WebMethod(exclude=true) Also, you can package a WSDL with your application to eliminate the need for the run time to generate a WSDL on your behalf.

Behavior change for exposing static and final methods:

Static or final methods in a service without an explicit SEI are no longer exposed as web services operations. To expose them, package the WSDL with the application and set `jaxws.runtime.legacyWebMethod=true`.

Procedure

1. Identify the methods that you want to expose as web services operations.
2. Review the rules for exposing methods as operations on classes annotated with the @WebService annotation.
3. Use the best practices for applying the @WebMethod and @WebService annotations in applications without SEIs to appropriately expose methods as operations within your web services.

Results

You have used the @WebMethod annotation to specify which methods to expose as web services operations.

Note:

If you have upgraded your application server environment and you are experiencing problems, review the following troubleshooting information.

Client errors indicate a mismatch between the WSDL file and portType when using a JAX-WS tooling version 2.1.6 or higher environment

You might receive a client-side error message like the following message:

```
javax.xml.ws.WebServiceException: The Endpoint validation failed to validate due to the following errors:
:: Invalid Endpoint Interface ::
:: The number of operations in the WSDL portType does not match the number of methods in the SEI or web service
 implementation class.  wsdl operations = [...] dispatch operations = [....]
```

To correct this problem, you must regenerate client artifacts to match the WSDL file.

Note: Be sure to regenerate your client side artifacts any time you receive an updated WSDL file.

Clients that perform a ?WSDL operation on web services have non-dispatchable operations

After performing a ?WSDL operation, you might receive a WSDL file that contains more operations than the JAX-WS runtime environment can dispatch. If the client tries to invoke any of the non-dispatchable operations, the client receives an error like the following message:

```
The endpoint reference (EPR) for the Operation not found is http://localhost:9086/example/BeanImpl2Service and the WSA
Action = <WSA_action_from_server>. If this EPR was previously reachable, contact the server administrator.
```

Clients must only access the operations that the web service intends to expose. You can correct this problem in one of the following ways:

- Modify the @WebMethod annotations in the web services application so that the resulting WSDL file exposes the correct set of operations.
- Set the `jaxws.runtime.legacyWebMethod` property to `false` to ensure that all operations in the WSDL are dispatched.

What to do next

Develop Java artifacts for JAX-WS applications from JavaBeans.

JAX-WS annotations

Java API for XML-Based Web Services (JAX-WS) relies on the use of annotations to specify metadata associated with web services implementations and to simplify the development of web services. Annotations describe how a server-side service implementation is accessed as a web service or how a client-side Java class accesses web services.

The JAX-WS programming standard introduces support for annotating Java classes with metadata that is used to define a service endpoint application as a web service and how a client can access the web service. JAX-WS supports the use of annotations based on the Metadata Facility for the Java Programming Language (Java Specification Request (JSR) 175) specification, the Web Services Metadata for the Java Platform (JSR 181) specification and annotations defined by the JAX-WS 2.0 and later (JSR 224) specification which includes JAXB annotations. Using annotations from the JSR 181 standard, you can simply annotate the service implementation class or the service interface and now the application is enabled as a web service. Using annotations within the Java source simplifies development and

deployment of web services by defining some of the additional information that is typically obtained from deployment descriptor files, WSDL files, or mapping metadata from XML and WSDL into the source artifacts.

Use annotations to configure bindings, handler chains, set names of portType, service and other WSDL parameters. Annotations are used in mapping Java to WSDL and schema, and at runtime to control how the JAX-WS runtime processes and responds to web service invocations.

For JAX-WS web services, the use of the `webservices.xml` deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

Starting with WebSphere Application Server Version 7.0 and later, Java EE 5 application modules (Web application modules version 2.5 or above, or EJB modules version 3.0 or above) are scanned for annotations to identify JAX-WS services and clients. However, pre-Java EE 5 application modules (web application modules version 2.4 or before, or EJB modules version 2.1 or before) are not scanned for JAX-WS annotations, by default, for performance considerations.

In the Version 6.1 Feature Pack for Web Services, the default behavior is to scan pre- Java Platform, Enterprise Edition (Java EE) 5 web application modules to identify JAX-WS services and to scan pre-Java EE 5 web application modules and EJB modules for service clients during application installation. Because the default behavior for WebSphere Application Server Version 7.0 and later is to not scan pre-Java EE 5 modules for annotations during application installation or server startup, to preserve backward compatability with the feature pack from previous releases, you must configure one of the following properties:

- You can set the `UseWSFEP61ScanPolicy` property in the META-INF/MANIFEST.MF of a WAR file or EJB module to `true`. For example:

```
Manifest-Version: 1.0
UseWSFEP61ScanPolicy: true
```

  When this property is set to `true` in the META-INF/MANIFEST.MF file of the module, the module is scanned for JAX-WS annotations regardless of the Java EE version of the module. The default value is `false` and when the default value is in effect, JAX-WS annotations are only supported in modules whose version is Java EE 5 or later.
- You can set the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` custom Java virtual machine (JVM) property using the administrative console. See the JVM custom properties documentation for the correct navigation path to use. To request annotation scanning in all modules regardless of their Java EE version, set the custom property `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` to `true`. You must change the setting on each server that requires a change in the default behavior.

If the property is set within the META-INF/MANIFEST.MF file of the module, this setting takes precedence over the server's custom JVM property. When using either property, you must establish the desired annotation scanning behavior before the application is installed. You cannot dynamically change the scanning behavior once an application is installed. If changes to the behavior are required after your application is installed, you must first uninstall the application, specify the desired scanning behavior using the appropriate property and then install the application again. When federating nodes that have the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` set to `true` in the configuration of the servers contained within the node, this property does not affect the deployment manager. You must set the property to `true` on the deployment manager before the node is federated to preserve the behavior as it was on the node before federation.

Note: If this JVM property is being used on the z/OS platform, it must be set in both the servant and control regions of the server.

Note: When federating nodes that have the `com.ibm.websphere.webservices.UseWSFEP61ScanPolicy` set to `true` in the configuration of the servers contained within the node, this does not affect the deployment manager. You must set the property to `true` on the deployment manager before the node is federated if you wish to preserve the behavior as it was on the node before federation.

Annotations supported by JAX-WS are listed in the table below. The target for annotations is applicable for these Java objects:

- types such as a Java class, enum or interface
- methods
- fields representing local instance variables within a Java class
- parameters within a Java method

Table 125. Web services Metadata Annotations (JSR 181). Describes the supported web services metadata annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.jws. WebService`	The **@WebService** annotation marks a Java class as implementing a Web service or marks a service endpoint interface (SEI) as implementing a web service interface. **Important:** • A Java class that implements a web service must specify either the @WebService or @WebServiceProvider annotation. Both annotations cannot be present. This annotation is applicable on a client or server SEI or a server endpoint implementation class. • If the annotation references an SEI through the `endpointInterface` attribute, the SEI must also be annotated with the @WebService annotation. • See the exposing methods in SEI-based JAX-WS web services information to learn about best practices for using the @WebService and @WebMethod annotations on a service endpoint implementation to specify Java methods that you want to expose as JAX-WS web services.	• Annotation target: Type • Properties: - **`name`** The name of the `wsdl:portType`. The default value is the unqualified name of the Java class or interface. (String) - **`targetNamespace`** Specifies the XML namespace of the WSDL and XML elements generated from the web service. The default value is the namespace mapped from the package name containing the web service. (String) - **`serviceName`** Specifies the service name of the web service: `wsdl:service`. The default value is the simple name of the Java class + *Service*. (String) - **`endpointInterface`** Specifies the qualified name of the service endpoint interface that defines the services' abstract web service contract. If specified, the service endpoint interface is used to determine the abstract WSDL contract. (String) - **`portName`** The `wsdl:portName`. The default value is WebService.name +*Port* . (String) - **`wsdlLocation`** Specifies the web address of the WSDL document defining the web service. The web address is either relative or absolute. (String)
`javax.jws. WebMethod`	The **@WebMethod** annotation denotes a method that is a web service operation. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: - **`operationName`** Specifies the name of the `wsdl:operation` matching this method. The default value is the name of Java method. (String) - **`action`** Defines the action for this operation. For SOAP bindings, this value determines the value of the SOAPAction header. The default value is the name of Java method. (String) - **`exclude`** Specifies whether to exclude a method from the web service. The default value is `false`. (Boolean)

Table 125. Web services Metadata Annotations (JSR 181) (continued). Describes the supported web services metadata annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.jws. Oneway`	The **@Oneway** annotation denotes a method as a web service one-way operation that only has an input message and no output message. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • There are no properties on the Oneway annotation.
`javax.jws. WebParam`	The **@WebParam** annotation customizes the mapping of an individual parameter to a web service message part and XML element. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Parameter • Properties: **- name** The name of the parameter. If the operation is remote procedure call (RPC) style and the `partName` attribute is not specified, then this is the name of the `wsdl:part` attribute representing the parameter. If the operation is document style or the parameter maps to a header, then `-name` is the local name of the XML element representing the parameter. This attribute is required if the operation is document style, the parameter style is `BARE`, and the mode is `OUT` or `INOUT`. (String) **- partName** Defines the name of `wsdl:part` attribute representing this parameter. This is only used if the operation is RPC style, or the operation is document style and the parameter style is `BARE`. (String) **- targetNamespace** Specifies the XML namespace of the XML element for the parameter. Applies only for document bindings when the attribute maps to an XML element. The default value is the `targetNamespace` for the web service. (String) **- mode** The value represents the direction the parameter flows for this method. Valid values are `IN`, `INOUT`, and `OUT`. (String) **- header** Specifies whether the parameter is in a message header rather than a message body. The default value is `false`. (Boolean)

Table 125. Web services Metadata Annotations (JSR 181) (continued). Describes the supported web services metadata annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.jws. WebResult`	The **@WebResult** annotation customizes the mapping of a return value to a WSDL part or XML element. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: **- name** Specifies the name of the return value as it is listed in the WSDL file and found in messages on the wire. For RPC bindings, this is the name of the `wsdl:part` attribute representing the return value. For document bindings, the `-name` parameter is the local name of the XML element representing the return value. The default value is `return` for RPC and DOCUMENT/WRAPPED bindings. The default value is the method name + `Response` for DOCUMENT/BARE bindings. (String) **- targetNamespace** Specifies the XML namespace for the return value. This parameter is only used if the operation is RPC style or if the operation is DOCUMENT style and the parameter style is BARE. (String) **- header** Specifies whether the result is carried in a header. The default value is `false`. (Boolean) **- partName** Specifies the part name for the result with RPC or DOCUMENT/BARE operations. The default value is `@WebResult.name`. (String)
`javax.jws. HandlerChain`	The **@HandlerChain** annotation associates the web service with an externally defined handler chain. You can only configure the server side handler by using the @HandlerChain annotation on the Service Endpoint Interface (SEI) or the server endpoint implementation class. Use one of several ways to configure a client side handler. You can configure a client side handler by using the @HandlerChain annotation on the generated service class or SEI. Additionally, you can programmatically register your own implementation of the HandlerResolver interface on the Service, or programmatically set the handler chain on the Binding object.	• Annotation target: Type • Properties: **- file** Specifies the location of the handler chain file. The file location is either an absolute java.net.URL in external form or a relative path from the class file. (String) **- name** Specifies the name of the handler chain in the configuration file. (String)

Table 125. Web services Metadata Annotations (JSR 181) (continued). Describes the supported web services metadata annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.jws. SOAPBinding`	The **@SOAPBinding** annotation specifies the mapping of the web service onto the SOAP message protocol. Apply this annotation to a type or methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class. The method level annotation is limited in what it can specify and is only used if the `style` property is `DOCUMENT`. If the method level annotation is not specified, the **@SOAPBinding** behavior from the type is used.	• Annotation target: Type or Method • Properties: **- `style`** Defines encoding style for messages sent to and from the web service. The valid values are `DOCUMENT` and `RPC`. The default value is `DOCUMENT`. (String) **- `use`** Defines the formatting used for messages sent to and from the web service. The default value is `LITERAL`. `ENCODED` is not supported. (String) **- `parameterStyle`** Determines whether the method's parameters represent the entire message body or whether parameters are elements wrapped inside a top-level element named after the operation. Valid values are `WRAPPED` or `BARE`. You can only use the `BARE` value with `DOCUMENT` style bindings. The default value is `WRAPPED`. (String)

Table 126. JAX-WS Annotations (JSR 224). Describes the supported JAX-WS annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.xml.ws. Action`	The **@Action** annotation specifies the WS-Addressing action that is associated with a web service operation. When you use this annotation with a particular method, and generate the corresponding WSDL document, the WS-Addressing Action extension attribute is added to the input and output elements of the WSDL operation that corresponds to that method. To add this attribute to the WSDL operation, you must also specify the @Addressing annotation on the server endpoint implementation class. If you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.	• Annotation target: Method • Properties: **- fault** Specifies the array of `FaultAction` for the `wsdl:fault` of the operation. (String) **- input** Specifies the action for the`wsdl:input` of the operation. (String) **- output** Specifies the action for the`wsdl:output` of the operation. (String)
`javax.xml.ws. BindingType`	The **@BindingType** annotation specifies the binding to use when publishing an endpoint of this type. Apply this annotation to a server endpoint implementation class. **Important:** You can use the @BindingType annotation on the JavaBeans endpoint implementation class to enable MTOM by specifying either `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING` or `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_MTOM_BINDING` as the value for the annotation.	• Annotation target: Type • Properties: **- value** Indicates the binding identifier web address. Valid values are `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`, `javax.xml.ws.soap.SOAPBinding.SOAP12HTTP_BINDING`, and `javax.xml.ws.http.HTTPBinding.HTTP2HTTP_BINDING`. The default value is `javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_BINDING`. (String)
`javax.xml.ws. FaultAction`	The **@FaultAction** annotation specifies the WS-Addressing action that is added to a fault response. This annotation must be contained within an @Action annotation. When you use this annotation with a particular method, the WS-Addressing FaultAction extension attribute is added to the fault element of the WSDL operation that corresponds to that method. To add this attribute to the WSDL operation, you must also specify the @Addressing annotation on the server endpoint implementation class. If you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.	• Annotation target: Method • Properties: **- value** Specifies the action of the `wsdl:fault` of the operation. (String) **- output** Specifies the name of the exception class. (String) **- className** Specifies the name of the class representing the request wrapper. (String)

Table 126. JAX-WS Annotations (JSR 224) (continued). Describes the supported JAX-WS annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.xml.ws. RequestWrapper`	The **@RequestWrapper** annotation supplies the JAXB generated request wrapper bean, the element name, and the namespace for serialization and deserialization with the request wrapper bean that is used at runtime. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the `className` attribute is required in this case. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: - **`localName`** Specifies the local name of the XML schema element representing the request wrapper. The default value is the `operationName` as defined in javax.jws.WebMethod annotation. (String) - **`targetNamespace`** Specifies the XML namespace of the request wrapper method. The default value is the target namespace of the SEI. (String) - **`className`** Specifies the name of the class representing the request wrapper. (String) - **`partName`** Specifies the name of the wsdl:part attribute that represents the XML schema element for the RequestWrapper class. This property is applicable for JAX-WS 2.2 and later. (String)
`javax.xml.ws. ResponseWrapper`	The **@ResponseWrapper** annotation supplies the JAXB generated response wrapper bean, the element name, and the namespace for serialization and deserialization with the response wrapper bean that is used at runtime. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the `className` attribute is required in this case. Apply this annotation to methods on a client or server Service Endpoint Interface (SEI) or a server endpoint implementation class.	• Annotation target: Method • Properties: - **`localName`** Specifies the local name of the XML schema element representing the request wrapper. The default value is the `operationName` + *Response*. `operationName` is defined in javax.jws.WebMethod annotation. (String) - **`targetNamespace`** Specifies the XML namespace of the request wrapper method. The default value is the target namespace of the SEI. (String) - **`className`** Specifies the name of the class representing the response wrapper. (String) - **`partName`** Specifies the name of the wsdl:part attribute that represents the XML schema element for the ResponseWrapper class. This property is applicable for JAX-WS 2.2 and later. (String)

Table 126. JAX-WS Annotations (JSR 224) (continued). Describes the supported JAX-WS annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.xml.ws. RespectBinding`	The **@RespectBinding** annotation specifies whether the JAX-WS implementation must use the contents of the `wsdl:binding` for an endpoint. When this annotation is specified, a check is performed to ensure all required WSDL extensibility elements with the enabled attribute set to `true` are supported. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Method • Properties: **- enabled** Specifies whether the `wsdl:binding` must be used or not. The default value is `true`. (Boolean)
`javax.xml.ws. ServiceMode`	The **@ServiceMode** annotation specifies whether a service provider needs to have access to an entire protocol message or just the message payload. **Important:** The @ServiceMode annotation is only supported on classes that are annotated with the @WebServiceProvider annotation.	• Annotation target: Type • Properties: **- value** Indicates whether the provider class accepts the payload of the message, `PAYLOAD` or the entire message `MESSAGE`. The default value is `PAYLOAD`. (String)
`javax.xml.ws. soap.Addressing`	The **@Addressing** annotation specifies that this service wants to enable WS-Addressing support. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Type • Properties: **- enabled** Specifies if WS-Addressing is enabled or not. The default value is `true`. (Boolean) **- required** Specifies that WS-Addressing headers must be present on incoming messages. The default value is `false`. (Boolean) **- responses** Specifies the message exchange pattern to use. The default value is `Responses.ALL`. This property is applicable for JAX-WS 2.2 and later. (String)

Table 126. JAX-WS Annotations (JSR 224) (continued). Describes the supported JAX-WS annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.xml.ws. soap.MTOM`	The **@MTOM** annotation specifies whether binary content in the body of a SOAP message is sent using MTOM. Apply this annotation to a service endpoint implementation class.	• Annotation target: Class • Properties: - **`enabled`** Specifies if MTOM is enabled for the JAX-WS endpoint. The default value is `true`. (Boolean) - **`threshold`** Specifies the minimum size for messages that are sent using MTOM. When the message size is less than this specified integer, the message is inlined in the XML document as base64 or hexBinary data. (integer)
`javax.xml.ws. WebFault`	The **@WebFault** annotation maps WSDL faults to Java exceptions. It is used to capture the name of the fault during the serialization of the JAXB type that is generated from a global element referenced by a WSDL fault message. It can also be used to customize the mapping of service specific exceptions to WSDL faults. This annotation can only be applied to a fault implementation class on the client or server.	• Annotation target: Type • Properties: - **`name`** Specifies the local name of the XML element that represents the corresponding fault in the WSDL file. The actual value must be specified. (String) - **`targetNamespace`** Specifies the namespace of the XML element that represents the corresponding fault in the WSDL file. (String) - **`faultBean`** Specifies the name of the fault bean class. (String) - **`messageName`** Specifies the name of the wsdl:message attribute that represents the corresponding fault in the WSDL file. This property is applicable for JAX-WS 2.2 and later. (String)

Table 126. JAX-WS Annotations (JSR 224) (continued). Describes the supported JAX-WS annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.xml.ws. WebServiceProvider`	The **@WebServiceProvider** annotation denotes that a class satisfies requirements for a JAX-WS Provider implementation class. **Important:** • A Java class that implements a web service must specify either the @WebService or @WebServiceProvider annotation. Both annotations cannot be present. • The @WebServiceProvider annotation is only supported on the service implementation class. Any class with the @WebServiceProvider annotation must implement the javax.xml.ws.Provider interface.	• Annotation target: Type • Properties: - **`targetNamespace`** Specifies the XML namespace of the WSDL and XML elements generated from the web service. The default value is the namespace mapped from the package name containing the web service. (String) - **`serviceName`** Specifies the service name of the web service: `wsdl:service`. The default value is the simple name of the Java class + *Service*. (String) - **`portName`** The `wsdl:portName`. The default value is the name of the class + *Port*. (String) - **`wsdlLocation`** The web address of the WSDL document defining the web service. This attribute is required. (String)

Table 126. JAX-WS Annotations (JSR 224) (continued). Describes the supported JAX-WS annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.xml.ws. WebServiceRef`	The **@WebServiceRef** annotation defines a reference to a web service invoked by the client. **Note:** • The @WebServiceRef annotation can be used to inject instances of JAX-WS services and ports. • The @WebServiceRef annotation is only supported in certain class types. Examples are JAX-WS endpoint implementation classes, JAX-WS handler classes, Enterprise JavaBeans classes, and servlet classes. This annotation is supported in the same class types as the @Resource annotation. See the Java Platform, Enterprise Edition (Java EE) 5 specification for a complete list of supported class types.	• Annotation target: Type, Field or Method • Properties: **- name** Specifies the JNDI name of the resource. The field name is the default for field annotations. The JavaBeans property name that corresponds to the method is the default for method annotations. You must specify a value for class annotations as there is no default. (String) **- type** Indicates the Java type of the resource. The field type is the default for field annotations. The type of the JavaBeans property is the default for method annotations. You must specify a value for class annotations as there is no default. (Class) **- mappedName** Specifies the name to map this resource to. (String) **- value** Indicates the value of the service class and it is a type that extends `javax.xml.ws.Service`. This attribute is required when the type of the reference is a service endpoint interface. (Class) **- wsdlLocation** The web address of the WSDL document defining the web service. This attribute is required. (String) **- lookup** Specifies the JNDI lookup name for the target web service. This property is applicable for JAX-WS 2.2 and later. (String)
`javax.xml.ws. WebServiceRefs`	The **@WebServiceRefs** annotation associates multiple @WebServiceRef annotations with a specific class. **Note:** The @WebServiceRef annotation is only supported in certain class types. Examples are JAX-WS endpoint implementation classes, JAX-WS handler classes, Enterprise JavaBeans classes, and servlet classes. This annotation is supported in the same class types as the @Resource annotation. See the Java Platform, Enterprise Edition (Java EE) 5 specification for a complete list of supported class types.	• Annotation target: Type • Properties: **- value** Specifies an array for multiple web service reference declarations. This attribute is required.

Table 127. JAX-WS Common Annotations (JSR 250). Describes the supported JAX-WS common annotations and their associated properties.

Annotation class	Annotation	Properties
`javax.annotation. Resource`	The **@Resource** annotation marks a WebServiceContext resource needed by the application. **Note:** Applying this annotation to a WebServiceContext type field on the server endpoint implementation class for a JavaBeans endpoint or a Provider endpoint results in the container injecting an instance of the WebServiceContext into the specified field. When this annotation is used in place of the @WebServiceRef annotation, the rules described for the @WebServiceRef annotation apply.	• Annotation target: Field or Method • Properties: **- type** Indicates the Java type of the resource. You are required to use the default, `java.lang.Object` or `javax.xml.ws.Web ServiceContext` value. If the type is the default, the resource must be injected into a field or a method. In this case, the type of the field or the type of the JavaBeans property defined by the method must be `javax.xml.ws.WebServiceContext`. (Class) If you are using this annotation to inject a web service, see the description of the @WebServiceRef `type` attribute.
`javax.annotation. Resources`	The **@Resources** annotation associates multiple @Resource annotations with a specific class and serves as a container for multiple resource declarations.	• Annotation target: Field or Method • Properties: **- value** Specifies an array for multiple @Resource annotations. This attribute is required.
`javax.annotation. PostConstruct`	The **@PostConstruct** annotation marks a method that needs to run after dependency injection is performed on the class. Apply this annotation to a JAX-WS application handler, a server endpoint implementation class.	• Annotation target: Method
`javax.annotation. PreDestroy`	The **@PreDestroy** annotation marks a method that must be run when the instance is in the process of being removed by the container. Apply this annotation to a JAX-WS application handler or a server endpoint implementation class.	• Annotation target: Method

Table 128. IBM proprietary annotations. Describes the supported IBM proprietary annotations and their associated properties.

Annotation class	Annotation	Properties
`com.ibm.websphere. wsaddressing. jaxws21. SubmissionAddressing`	The **@SubmissionAddressing** annotation specifies that this service wants to enable WS-Addressing support for the 2004/08 WS-Addressing specification. This annotation is part of the IBM implementation of the JAX-WS 2.1 specification. Apply this annotation to methods on a server endpoint implementation class.	• Annotation target: Type • Properties: - **`enabled`** Specifies if WS-Addressing is enabled or not. The default value is `true`. (Boolean) - **`required`** Specifies that WS-Addressing headers must be present on incoming messages. The default value is `false`. (Boolean)

Generating Java artifacts for JAX-WS applications

Use Java API for XML-Based Web Services (JAX-WS) tools to generate the necessary JAX-WS and Java Architecture for XML Binding (JAXB) Java artifacts that are needed for JAX-WS web services applications when starting from JavaBeans or enterprise beans components.

Before you begin

To develop a Java API for XML-Based Web Services (JAX-WS) web service application, you must first develop a service endpoint interface (SEI) implementation that explicitly or implicitly describes the SEI.

About this task

When using a bottom-up approach to develop JAX-WS web services, use the **wsgen** command-line tool on the existing service endpoint implementation. The **wsgen** tool processes a compiled service endpoint implementation class as input and generates the following portable artifacts:

- Java Architecture for XML Binding (JAXB) classes that are required to marshal and unmarshal the message contents.
- a Web Services Description Language (WSDL) file if the optional *-wsdl* argument is specified.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

You are not required to develop a WSDL file when developing JAX-WS Web services using the bottom-up approach of starting with JavaBeans. The use of annotations provides all of the WSDL information necessary to configure the service endpoint or the client. The application server supports WSDL 1.1 documents that comply with Web Services-Interoperability (WS-I) Basic Profile 1.1 specifications and are either Document/Literal style documents or RPC/Literal style documents. Additionally, WSDL documents with bindings that declare a `USE` attribute of value `LITERAL` are supported while the value, `ENCODED`, is not supported. For WSDL documents that implement a Document/Literal wrapped pattern, a root element is declared in the XML schema and is used as an operation wrapper for a message flow. Separate wrapper element definitions exist for both the request and the response.

To ensure the **wsgen** command does not miss inherited methods on a service endpoint implementation bean, you must either add the @WebService annotation to the desired superclass or you can override the inherited method in the implementation class with a call to the superclass method.

Although a WSDL file is typically optional when developing a JAX-WS service implementation bean, it is required if your JAX-WS endpoints are exposed using the SOAP over JMS transport and you are publishing your WSDL file. If you are developing an enterprise JavaBeans service implementation bean that is invoked using the SOAP over JMS transport, and you want to publish the WSDL so that the

published WSDL file contains the fully resolved JMS endpoint URL, then have **wsgen** tool generate the WSDL file by specifying the *-wsdl* argument. In this scenario, you must package the WSDL file with your web service application.

In addition to using the tools from the command-line, you can invoke these JAX-WS tools from within the Ant build environments. Use the `com.sun.tools.ws.ant.WsGen` Ant task from within the Ant build environment to invoke the wsgen tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Procedure

1. Locate your service endpoint implementation class file.
2. Run the **wsgen** command to generate the portable artifacts. The **wsgen** tool is located in the *app_server_root*/bin/ directory.

   (Optional) Use the following options with the **wsgen** command:
   - Use the **-verbose** option to see a list of generated files along with additional informational messages.
   - Use the **-keep** option to keep generated Java files.
   - Use the **-wsdl** option to generate a WSDL file. If you are developing a service implementation bean that will be invoked using the HTTP transport, then the WSDL file generated by the **wsgen** command-line tool during this step is optional. However, if you are developing a service implementation bean that will be invoked using the SOAP over JMS transport, then the WSDL file generated by the **wsgen** tool during this step is required in subsequent developing JAX-WS applications steps, so it is not optional.

   Read about **wsgen** to learn more about this command and additional options that you can specify.

Results

You have the required Java artifacts to create a JAX-WS web service.

Note: The **wsgen** command does not differentiate the XML namespace between multiple XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates that you have classes or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

With JAX-WS applications, the **wsgen** command-line tool might not locate shared class files. You can specify the location of these class files using the com.ibm.websphere.webservices.wsdl_generation_extra_classpath custom property. For more information, see the documentation about the Java virtual machine custom properties.

Example

The following example demonstrates how to use the **wsgen** command to process the service endpoint implementation class to generate JAX-WS artifacts. This example EchoService service implementation class uses an explicit JavaBeans service endpoint.

1. Copy the sample `EchoServicePortTypeImpl` service implementation class file and the associated `EchoServicePortType` service interface class file into a directory. The directory must contain a directory tree structure that corresponds to the `com.ibm.was.wssample.echo` package name for the class file.

```
/* This is a sample EchoServicePortTypeImpl.java file.    */
package com.ibm.was.wssample.echo;
```

```
@javax.jws.WebService(serviceName = "EchoService", endpointInterface =
"com.ibm.was.wssample.echo.EchoServicePortType",
targetNamespace="http://com/ibm/was/wssample/echo/",
portName="EchoServicePort")

public class EchoServicePortTypeImpl implements EchoServicePortType {

 public EchoServicePortTypeImpl() {
 }

 public String invoke(String obj) {
  System.out.println(">> JAXB Provider Service:
Request received.\n");
  String str = "Failed";
  if (obj != null) {
   try {
    str = obj;
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
  return str;

 }

}
```

```
/* This is a sample EchoServicePortType.java file.    */
package com.ibm.was.wssample.echo;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;


@WebService(name = "EchoServicePortType", targetNamespace =
"http://com/ibm/was/wssample/echo/",
 wsdlLocation="WEB-INF/wsdl/Echo.wsdl")

public interface EchoServicePortType {


    /**
     *
     * @param arg0
     * @return
     *     returns java.lang.String
     */
    @WebMethod
    @WebResult(name = "response", targetNamespace =
    "http://com/ibm/was/wssample/echo/")
    @RequestWrapper(localName = "invoke", targetNamespace =
    "http://com/ibm/was/wssample/echo/",
     className = "com.ibm.was.wssample.echo.Invoke")
    @ResponseWrapper(localName = "echoStringResponse",
     targetNamespace = "http://com/ibm/was/wssample/echo/",
     className = "com.ibm.was.wssample.echo.EchoStringResponse")
    public String invoke(
        @WebParam(name = "arg0", targetNamespace =
        "http://com/ibm/was/wssample/echo/")
        String arg0);

}
```

2. Run the **wsgen** command from the *app_server_root*\bin\ directory. The `-cp` option specifies the location of the service implementation class file. The `-s` option specifies the directory for the generated source files. The `-d` option specifies the directory for the generated output files. When using the `-s` or `-d` options, you must first create the directory for the generated output files.

After generating the Java artifacts using the **wsgen** command, the following files are generated:

```
/generated_source/com/ibm/was/wssample/echo/EchoStringResponse.java
/generated_source/com/ibm/was/wssample/echo/Invoke.java
/generated_artifacts/EchoService.wsdl
/generated_artifacts/EchoService_schema1.xsd
/generated_artifacts/com/ibm/was/wssample/echo/EchoStringResponse.class
/generated_artifacts/com/ibm/was/wssample/echo/Invoke.class
```

The EchoStringResponse.java and Invoke.java files are the generated Java class files. The compiled versions of the generated Java files are EchoStringResponse.class and Invoke.class files. The EchoService.wsdl and EchoService_schema1.xsd files are generated because the `-wsdl` option was specified.

What to do next

Complete the implementation of your JAX-WS web service application.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
: Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

  The corresponding product variable is *smpe.install.root*.

  The default is /usr/lpp/zWebSphere/V8R5.

wsgen command for JAX-WS applications

The **wsgen** command-line tool generates the necessary artifacts required for Java API for XML Web Services (JAX-WS) applications when starting from Java code. The generated artifacts are Java 5 compliant, making them portable across different Java versions and platforms.

When using a bottoms-up approach to develop JAX-WS web services and you are starting from a service endpoint implementation, use the **wsgen** tool to generate the required JAX-WS artifacts.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

The **wsgen** tool accepts a properly annotated service endpoint implementation using the @WebService annotation as input and generates the following artifacts:

- any additional Java Architecture for XML Binding (JAXB) classes that are required to marshal and unmarshal the message contents.
- a WSDL file if the optional `-wsdl` argument is specified. The **wsgen** tool does not automatically generate the WSDL file.

When using JAX-WS V2.2 tools, java.lang.RuntimeException and java.rmi.RemoteException references and their subclasses are no longer mapped in the WSDL file. This behavior change complies with the JAX-WS V2.1 specification conformance rule that is described in section 3.7 of the specification. This conformance rule specifies that the java.lang.RuntimeException and java.rmi.RemoteException classes and their subclasses cannot be treated as service-specific exceptions and mapped in the WSDL file.

In addition to using the tools from the command line, you can invoke these JAX-WS tools from within the Ant build environments. Use the `com.sun.tools.ws.ant.WsGen` Ant task from within the Ant build environment to invoke the wsgen tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Note: The **wsgen** command does not differentiate the XML namespace between multiple XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

With JAX-WS applications, the **wsgen** command-line tool might not locate shared class files. You can specify the location of these class files using the com.ibm.websphere.webservices.wsdl_generation_extra_classpath custom property. For more information, see the documentation about the Java virtual machine custom properties.

Syntax

The command-line syntax is:

Parameters

The *service_implementation_class* name is the only parameter that is required. The following parameters are optional for the **wsgen** command:

-classpath <path>
Specifies the location of the service implementation class.

-cp <path>
Specifies the location of the service implementation class. This parameter is the same as *-classpath <path>*.

-d <directory>
Specifies where to place the generated output files.

-extension
Specifies whether to enable custom extensions for functionality not specified by the JAX-WS specification. Use of the extensions can result in applications that are not portable or do not interoperate with other implementations.

-help
Displays the help menu.

-keep
Specifies whether to keep the generated source files.

-r <directory>
This parameter is only used with the `-wsdl` parameter. Specifies where to place the generated WSDL file.

-s <directory>
Specifies the directory to place the generated source files.

-verbose
Specifies to output messages about what the compiler is doing.

-version
Prints the version information. If you specify this option, only the version information is output and normal command processing does not occur.

-wsdl [:protocol]
By default, the **wsgen** tool does not generate a WSDL file. This optional parameter causes **wsgen** to generate a WSDL file and is typically only used to enable a developer to review a WSDL file before the endpoint is deployed. The `protocol` is optional and specifies the protocol used in the `wsdl:binding`. Valid values for `protocol` are `soap 1.1` and `Xsoap 1.2`. The default value is `soap 1.1`. The `Xsoap 1.2` value is not standard and is only used with the *-extension* option.

-servicename <name>
This parameter is only used with the `-wsdl` option. Specifies a `wsdl:service` name to be generated in the WSDL file. For example,

```
-service name "{http://mynamespace/}MyService"
```

-portname
This parameter is only used with the `-wsdl` option. Specifies a `wsdl:port` name to be generated in the WSDL file. For example,

```
-portname "{http://mynamespace/}MyPort"
```

Mapping between Java language, WSDL and XML for JAX-WS applications

Data for Java API for XML Web Services (JAX-WS) applications flows as extensible Markup Language (XML). JAX-WS applications use mappings to describe the data conversion between the Java language and extensible Markup Language (XML) technologies, including XML Schema, Web Services Description Language (WSDL) and SOAP that are supported by the application server.

For web services based on the JAX-WS programming model, mappings between the Java language and XML are specified by the JAX-WS specification and the Java Architecture for XML Binding (JAXB) specification for data bindings. JAX-WS leverages the JAXB API and tools as the binding technology for mappings between Java objects and XML documents. JAX-WS tooling relies on JAXB tooling for default data binding for two-way mappings between Java objects and XML documents.

The JAX-WS specification describes the mapping between Web Services Description Language (WSDL) files and the Java language. The supported mappings include WSDL-to-Java mappings and Java-to-WSDL mappings. WSDL 1.1 is required by the JAX-WS 2.0 specification. You can use annotations to customize the mapping from Java artifacts to their associated WSDL components. Refer to the JAX-WS specification for details describing the WSDL-to-Java mappings and Java-to-WSDL mappings.

Data binding mappings used by the JAX-WS programming model are described by the JAXB specification. Refer to the JAXB specification for details that describe the JAXB mappings for the Java representation of XML content, including the default and custom bindings between XML schema to Java representations.

Enabling MTOM for JAX-WS web services

With Java API for XML-Based Web Services (JAX-WS), you can send binary attachments such as images or files along with web services requests. JAX-WS adds support for optimized transmission of binary data as specified by the SOAP Message Transmission Optimization Mechanism (MTOM) specification.

About this task

JAX-WS supports the use of SOAP Message Transmission Optimized Mechanism (MTOM) for sending binary attachment data. By enabling MTOM, you can send and receive binary data optimally without incurring the cost of data encoding needed to embed the binary data in an XML document.

The application server supports sending attachments using MTOM only for JAX-WS applications. This product also provides the ability to provide attachments with Web Services Security SOAP messages by using the new MTOM and XOP standards.

JAX-WS applications can send binary data as base64 or hexBinary encoded data contained within the XML document. However, to take advantage of the optimizations provided by MTOM, enable MTOM to send binary base64 data as attachments contained outside the XML document. MTOM optimization is not enabled by default. JAX-WS applications require separate configuration of both the client and the server artifacts to enable MTOM support.

Procedure

1. Develop Java artifacts for your JAX-WS application that includes an XML schema or Web Services Description Language (WSDL) file that represents your web services application data that includes a binary attachment.
   a. If you are starting with a WSDL file, develop Java artifacts from a WSDL file by using the wsimport command to generate the required JAX-WS portable artifacts.
   b. If you are starting with JavaBeans components, develop Java artifacts for JAX-WS applications and optionally generate a WSDL file using the wsgen command. The XML schema or WSDL file includes a `xsd:base64Binary` or `xsd:hexBinary` element definition for the binary data.
   c. You can also include the `xmime:expectedContentTypes` attribute on the element to affect the mapping by JAXB.
2. Enable MTOM on your endpoint implementation class using one of the following methods:
   - Use the @MTOM annotation on the endpoint.

     To enable MTOM on an endpoint, use the @MTOM (javax.xml.ws.soap.MTOM) annotation on the endpoint. The @MTOM annotation has two parameters, `enabled` and `threshold`. The `enabled` parameter has a boolean value and indicates if MTOM is enabled for the JAX-WS endpoint. The `threshold` parameter has an integer value, that must be greater than or equal to zero. When MTOM is enabled, any binary data whose size, in bytes, exceeds the threshold value is XML-binary Optimized Packaging (XOP) encoded or sent as an attachment. When the message size is less than the threshold value, the message is inlined in the XML document as either base64 or hexBinary data.

The following example snippet illustrates adding the @MTOM annotation so that MTOM is enabled for the JAX-WS `MyServiceImpl` endpoint and specifies a threshold value of 2048 bytes:

```
@MTOM(enabled=true, threshold=2048)
@WebService
public class MyServiceImpl {
...
}
```

Additionally, you can use the @BindingType (javax.xml.ws.BindingType) annotation on an endpoint implementation class to specify that the endpoint supports one of the MTOM binding types so that the response messages are MTOM-enabled. The javax.xml.ws.SOAPBinding class defines two different constants, `SOAP11HTTP_MTOM_BINDING` and `SOAP12HTTP_MTOM_BINDING` that you can use for the value of the @BindingType annotation; for example:

```
// This example is for SOAP version 1.1.
@BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING)
@WebService
public class MyServiceImpl {
...
}
```

```
// This example is for SOAP version 1.2.
@BindingType(value = SOAPBinding.SOAP12HTTP_MTOM_BINDING)
@WebService
public class MyServiceImpl {
...
}
```

The presence and value of an @MTOM annotation overrides the value of the @BindingType annotation. For example, if the @BindingType indicates MTOM is enabled, but an @MTOM annotation is present with an `enabled` value of `false`, then MTOM is not enabled.

- Use the `<enable-mtom>` and `<mtom-threshold>` deployment descriptor elements.

  You can use the `<enable-mtom>` and `<mtom-threshold>` elements within the `<port-component>` element in the webservices.xml deployment descriptor as an alternative to using the @MTOM annotation on the service endpoint implementation class; for example:

```
<port-component>
   <port-component-name>MyPort1</port-component-name>
   <enable-mtom>true</enable-mtom>
   <mtom-threshold>2048</mtom-threshold>
   <service-impl-bean>
    <servlet-link>MyPort1ImplBean</servlet-link>
   </service-impl-bean>
</port-component>
```

Note: The deployment descriptor elements take precedence over the corresponding attributes in the MTOM annotation. For example, if the `enabled` attribute is set to `true` in the annotation, but the `<enable-mtom>` element is set to `false` in the `webservices.xml` file, MTOM is not enabled for the corresponding endpoint.

3. Enable MTOM on your client to optimize the binary messages that are sent from the client to the server. Use one of the following methods to enable MTOM on your client:

   - Enable MTOM on a Dispatch client.

     The following examples use SOAP version 1.1.

     The first method uses SOAPBinding.setMTOMEnabled(); for example:

```
SOAPBinding binding = (SOAPBinding)dispatch.getBinding();
    binding.setMTOMEnabled(true);
```

     The second method uses Service.addPort; for example:

```
    Service svc = Service.create(serviceName);
    svc.addPort(portName,SOAPBinding.SOAP11HTTP_MTOM_BINDING,endpointUrl);
```

     The third method uses MTOMFeature; for example:

```
   MTOMFeature mtom = new MTOMFeature(true, 2048);
    Service svc = Service.create(serviceName);
    svc.addPort(portName, SOAPBinding.SOAP11_HTTP_BINDING, endpointUrl);
    Dispatch<Source> dsp = svc.createDispatch(portName, Source.class, Service.Mode.PAYLOAD, mtom);
```

   - Enable MTOM on a Dynamic Proxy client.

```
// Create a BindingProvider bp from a proxy port.
  Service svc = Service.create(serviceName);
  MtomSample proxy = svc.getPort(portName, MtomSample.class);
  BindingProvider bp = (BindingProvider) proxy;
```

```
  //Enable MTOM using the SOAPBinding.
  MtomSample proxy = svc.getPort(portName, MtomSample.class);
  BindingProvider bp = (BindingProvider) proxy;
  SOAPBinding binding = (SOAPBinding) bp.getBinding();
  binding.setMTOMEnabled(true);

  //Or, you can enable MTOM with the MTOMFeature.
  MTOMFeature mtom = new MTOMFeature();
  MtomSample proxy = svc.getPort(portName, MtomSample.class, mtom);
```

- Enable MTOM on your client using the @MTOM annotation; for example:

```
public class MyClientApplication {

    // Enable MTOM for a port-comonent-ref resource injection.
     @MTOM(enabled=true, threshold=1024)
     @WebServiceRef(MyService.class)
     private MyPortType myPort;
    ...
}
```

- Enable MTOM on your client using deployment descriptor elements within a port-component-ref element; for example:

```
<service-ref>
    <service-ref-name>service/MyPortComponentRef</service-ref-name>
    <service-interface>com.example.MyService</service-ref-interface>
    <port-component-ref>
        <service-endpoint-interface>com.example.MyPortType</service-endpoint-interface>
        <enable-mtom>true</enable-mtom>
        <mtom-threshold>1024</mtom-threshold>
    </port-component-ref>
</service-ref>
```

Note: The deployment descriptor elements take precedence over the corresponding attributes in the MTOM annotation. For example, if the `enabled` attribute is set to `true` in the annotation, but the `<enable-mtom>` element is set to `false` in the deployment descriptor entry for the service-ref of the client, MTOM is not enabled for that service reference.

Results

You have developed a JAX-WS web services server and client application that optimally sends and receives binary data using MTOM.

Example

The following example illustrates enabling MTOM support on both the web services client and server endpoint when starting with an WSDL file.

1. Locate the WSDL file containing an `xsd:base64Binary` element. The following example is a portion of a WSDL file that contains an `xsd:base64Binary` element.

```
  <types>
    ........
    <xs:complexType name="ImageDepot">
        <xs:sequence>
            <xs:element name="imageData"
                        type="xs:base64Binary"
                        xmime:expectedContentTypes="image/jpeg"/>

        </xs:sequence>
    </xs:complexType>
    ........
</types>
```

2. Run the **wsimport** command from the *app_server_root*\bin\ directory against the WSDL file to generate a set of JAX-WS portable artifacts.

   Depending on the `expectedContentTypes` value contained in the WSDL file, the JAXB artifacts generated are in the Java type as described in the following table:

Table 129. Mapping of MIME type and Java type. Describes the mapping between MIME types and Java types.

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String

Table 129. Mapping of MIME type and Java type (continued). Describes the mapping between MIME types and Java types.

MIME Type	Java Type
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
/	javax.activation.DataHandler

3. Use the JAXB artifacts in the same manner as in any other JAX-WS application. Use these beans to send binary data from both the Dispatch and the Dynamic Proxy client APIs.
4. Enable MTOM on a Dispatch client.

```
//Create the Dispatch instance.
        JAXBContext jbc = JAXBContext.newInstance("org.apache.axis2.jaxws.sample.mtom");
        Dispatch<Object> dispatch = svc.createDispatch(portName, jbc, Service.Mode.PAYLOAD);

//Enable MTOM.
        SOAPBinding binding = (SOAPBinding) dispatch.getBinding();
        binding.setMTOMEnabled(true);
```

5. Enable MTOM on a Dynamic Proxy client.

```
//Create the Dynamic Proxy instance.
        Service svc = Service.create(serviceName);
        MtomSample proxy = svc.getPort(portName, MtomSample.class);

//Enable MTOM.
        BindingProvider bp = (BindingProvider) proxy;
        SOAPBinding binding = (SOAPBinding) bp.getBinding();
        binding.setMTOMEnabled(true);
```

   Now that you have enabled the JAX-WS client for MTOM, messages sent to the server have MTOM enabled. However, for the server to respond back to the client using MTOM, you must enable MTOM on the endpoint.
6. Enable MTOM on the endpoint implementation class.

```
@MTOM(enabled, threshold=4096)
@WebService (endpointInterface="org.apache.axis2.jaxws.sample.mtom.MtomSample")

public class MtomSampleService implements MtomSample {
   ....
}
```

   The `jaxax.xml.ws.SOAPBinding` class has a static member for each of the supported binding types. Include either the `SOAP11HTTP_MTOM_BINDING` or the `SOAP12HTTP_MTOM_BINDING` as the value for the `@BindingType` annotation. This value enables all server responses to have MTOM enabled.

When you enable MTOM on the server and the client, the binary data that represents the attachment is included as a Multipurpose Internet Mail Extensions (MIME) attachment to the SOAP message. Without MTOM, the same data is encoded in the format that describes the XML schema, either base64 or hexadecimal encoding, and included inline in the XML document.

This example illustrates an MTOM enabled SOAP version 1.1 message with an attachment. The `type` and `content-type` attributes both have the value, `application/xop+xml`, which indicates that the message was successfully optimized using XML-binary Optimized packaging (XOP) when MTOM was enabled. This example demonstrates how the optimized message looks on the wire with MTOM enabled.

```
... other transport headers ...
Content-Type: multipart/related; boundary=MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812;
type="application/xop+xml"; start="
<0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>"; start-info="text/xml"; charset=UTF-8

--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: application/xop+xml; charset=UTF-8; type="text/xml";
content-transfer-encoding: binary
content-id:
   <0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>

<?xml version="1.0" encoding="UTF-8"?>
         <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
            <soapenv:Header/>
            <soapenv:Body>
               <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
                  <input>
```

```
                    <imageData>
                       <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:1.urn:uuid:0FE43E4D025F0BF3DC11582467646811@apache.org"/>
                    </imageData>
                 </input>
              </sendImage>
           </soapenv:Body>
        </soapenv:Envelope>
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: text/plain
content-transfer-encoding: binary
content-id:
        <1.urn:uuid:0FE43E4D025F0BF3DC115824676468l1@apache.org>

... binary data goes here ...
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812--
```

In contrast, this example demonstrates a SOAP version 1.1 message on the wire without MTOM enabled. The binary data is included in the body of the SOAP message, and the SOAP message is not optimized.

```
... other transport headers ...
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Header/>
      <soapenv:Body>
         <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
            <input>
               <imageData>R0lGAD1 ... more base64 encoded data ... KTJk8giAAA7</imageData>
            </input>
         </sendImage>
      </soapenv:Body>
   </soapenv:Envelope>
```

For additional information, refer to the Samples section of the Information Center which includes a sample that demonstrates the use of MTOM with JAX-WS web services.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root

Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root

Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root

Refers to the installation root directory for Web Server Plug-ins.

profile_root

Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Enforcing adherence to WSDL bindings in JAX-WS web services

Java API for XML-Based Web Services (JAX-WS) Version 2.1 introduced the concept of *features* as a way to programmatically control specific functions and behaviors. The RespectBindingFeature is one of the supported standard features. You can use the RespectBindingFeature to control whether a JAX-WS implementation is required to respect the contents of a Web Services Description Language (WSDL) binding that is associated with an endpoint.

About this task

While WSDL documents are often used during the development process, the actual enforcement of the use of the WSDL document specifications, when they are provided, at run time has not been well defined in versions of the JAX-WS specification previous to Version 2.1. The JAX-WS Version 2.1 specification added the feature, RespectBindingFeature, to clarify the impact of the wsdl:binding in a JAX-WS runtime environment.

Enabling the feature, RespectBindingFeature, causes the JAX-WS runtime environment to inspect the wsdl:binding for an endpoint at run time to ensure that the binding of service endpoint interface (SEI) parameters and return values is respected. Additionally, this feature ensures that all required wsdl:binding extensions are either understood and used by the runtime environment, or the extensions have been explicitly disabled by the application. Your JAX-WS application can disable a specific wsdl:binding extension that has a defined WebServiceFeature interface by using the appropriate annotation that is associated with that feature, using an API that accepts the javax.xml.ws.WebServiceFeature interface, or configuring the deployment descriptors.

When the RespectBindingFeature feature is not enabled, which is the default, the runtime environment can choose whether any part of the wsdl:binding is enforced.

Procedure

1. Develop Java artifacts for your JAX-WS application that includes a Web Services Description Language (WSDL) file that represents your web services application.
   a. If you are starting with a WSDL file, develop Java artifacts from a WSDL file by using the wsimport command to generate the required JAX-WS portable artifacts.
   b. If you are starting with JavaBeans components, develop Java artifacts for JAX-WS applications and generate a WSDL file using the wsgen command.
2. If you want to enable RespectBindingFeature on your endpoint implementation class, use one of the following methods:
   - Use the @RespectBinding annotation on the endpoint.

     To enable RespectBinding on an endpoint, use the @RespectBinding (javax.xml.ws.RespectBinding) annotation on the endpoint. The @RespectBinding annotation has only one parameter, enabled. This parameter is optional. The enabled parameter has a Boolean value and indicates if RespectBindingFeature is enabled for the JAX-WS endpoint.

     The following example snippet illustrates adding the @RespectBinding annotation for the JAX-WS MyServiceImpl endpoint:

```
@RespectBinding(enabled=true))
@WebService
public class MyServiceImpl {
...
}
```

- Use the <respect-binding> deployment descriptor element.

  You can use the <respect-binding> element within the <port-component> element in the webservices.xml deployment descriptor as an alternative to using the @RespectBinding annotation on the service endpoint implementation class; for example:

```
<port-component>
     <port-component-name>MyPort1</port-component-name>
     <respect-binding>
         <enabled>true</enabled>
     </respect-binding>
     <service-impl-bean>
         <servlet-link>MyPort1ImplBean</servlet-link>
     </service-impl-bean>
</port-component>
```

3. If you want to enable RespectBindingFeature on your client, use one of the following methods:
   - Enable RespectBindingFeature on a dispatch client; for example:

```
RespectBindingFeature respectBinding = new RespectBindingFeature();
Service svc = Service.create(serviceName);
svc.addPort(portName, SOAPBinding.SOAP11_HTTP_BINDING, endpointUrl);
Dispatch<Source> dsp = svc.createDispatch(portName, Source.class, Service.Mode.PAYLOAD, respectBinding);
```

   - Enable RespectBindingFeature on a dynamic proxy client; for example:

```
// Create a dynamic proxy with RespectBinding enabled.
  Service svc = Service.create(serviceName);
  RespectBindingFeature respectBinding = new RespectBindingFeature();
  RespectBindingSample proxy = svc.getPort(portName, RespectBindingSample.class, respectBinding);
```

   - Enable RespectBindingFeature on your client using the @RespectBinding annotation; for example:

```
public class MyClientApplication {

               // Enable RespectBinding for a port-component-ref resource injection.
                   @RespectBinding(enabled=true)
                   @WebServiceRef(MyService.class)
                   private MyPortType myPort;
               ...
}
```

   - Enable RespectBindingFeature on your client using deployment descriptor elements within a port-component-ref element; for example:

```
<service-ref>
    <service-ref-name>service/MyPortComponentRef</service-ref-name>
    <service-interface>com.example.MyService</service-ref-interface>
    <port-component-ref>
        <service-endpoint-interface>com.example.MyPortType</service-endpoint-interface>
        <respect-binding>
            <enabled>true</enabled>
        </respect-binding>
    </port-component-ref>
</service-ref>
```

Results

By implementing the feature, RespectBindingFeature, you have specified to enforce adherence of the contents of a WSDL binding that is associated with an endpoint for your JAX-WS application.

Developing a webservices.xml deployment descriptor for JAX-WS applications

Deployment descriptors are standard text files, formatted using XML and packaged in a web services application. You can optionally use the webservices.xml deployment descriptor to augment or override application metadata specified in annotations within Java API for XML-Based Web Services (JAX-WS) web services.

About this task

Similar to Java API for XML-based RPC (JAX-RPC) Web services, you can use deployment descriptors to describe JAX-WS web services. For JAX-WS web services, the use of the webservices.xml deployment

descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

A JAX-WS web service requires that you annotate your Java class with the `javax.jws.WebService` annotation or the `javax.jws.WebServiceProvider` annotation for Provider endpoints. You can use server-side deployment descriptors to override corresponding attributes of the annotation or to enhance information in annotations. There is a defined relationship between the deployment descriptor elements and the @WebService and @WebServiceProvider annotations. Refer to section 5.3 in the Web Services for Java Platform, Enterprise Edition (Java EE) specification, Version 1.2 for detailed information regarding the deployment descriptor elements and the mapping to the @WebService and @WebServiceProvider annotation attributes. There are also elements in the `webservice.xml` deployment descriptor that map to other annotations. For example, the deployment descriptor element <protocol-binding> maps to the @BindingType annotation, and the deployment descriptor element <enable-mtom> maps to the @MTOM annotation. For more information regarding the web services deployment descriptor elements, see section 7.1 in the Web Services for Java Platform, Enterprise Edition (Java EE) specification.

Procedure

Use assembly tools to generate the `webservice.xml` deployment descriptor.

Results

You have deployment descriptor templates that you can use to override JAX-WS annotation attributes or specify attributes that are not defined by the annotation.

Example

In the following example, the service implementation class for a JAX-WS web service includes the @WebService annotation:

```
@WebService(wsdlLocation="http://myhost.com/location/of/the/wsdl/ExampleService.wsdl")
```

The associated `webservices.xml` deployment descriptor specifies a different filename for the WSDL document as follows:

```
<webservices>
<webservice-description>
<webservice-description-name>ExampleService</webservice-description-name>
<wsdl-file>META-INF/wsdl/ExampleService.wsdl</wsdl-file>
...
</webservice-description>
</webservices>
```

The value that is specified in the deployment descriptor, `META-INF/wsdl/ExampleService.wsdl`, overrides the annotation value.

What to do next

Configure the `webservice.xml` deployment descriptor. After you configure the deployment descriptors, you must assemble the Web services application for deployment.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
Refers to the installation root directory for Web Server Plug-ins.

profile_root
Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Completing the JavaBeans implementation for JAX-WS applications

After you have developed the Java artifacts necessary to develop a Java API for XML-Based Web Services (JAX-WS) web service, you must complete the JavaBeans implementation to assemble a web application archive (WAR) file. The resulting WAR file contains the JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Generate Java artifacts for JAX-WS applications and optionally generate a WSDL file using the **wsgen** command-line tool. You can also optionally use deployment descriptors to augment or override binding information contained in annotations for JAX-WS web services.

About this task

For JAX-WS applications, complete the JavaBeans implementation by writing your business application.

Procedure

1. Write the JavaBeans implementation. The JavaBeans implementation is not generated by JAX-WS tooling.
2. Compile all the Java classes.

Results

You have now written your JavaBeans implementation to complete your web service application.

What to do next

After completing the JavaBeans implementation, assemble your web services application.

Completing the EJB implementation for JAX-WS applications

After you have developed the Java artifacts necessary to develop a Java API for XML-Based Web Services (JAX-WS) web service, you must complete the Enterprise JavaBeans (EJB) implementation to assemble a Java archive (JAR) file. The resulting JAR file contains the Enterprise JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Generate Java artifacts for JAX-WS applications and optionally generate a WSDL file using the **wsgen** command-line tool. You can also optionally use deployment descriptors to augment or override binding information contained in annotations for JAX-WS web services.

About this task

For JAX-WS applications, complete the enterprise beans implementation by writing your business application.

Procedure

1. Write the enterprise beans implementation. The enterprise beans implementation is not generated by JAX-WS tooling.
2. Compile all the Java classes.

Results

You have now written your enterprise beans implementation to complete your web service application.

What to do next

After completing the enterprise beans implementation, assemble your web services application.

Developing JAX-WS web services with WSDL files (top-down)

Setting up a development environment for web services

The application server provides command-line tools to develop web services clients and implementations that are based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. You must set up your development environment before you start developing web services.

Before you begin

Before you can set up a web services development environment within WebSphere Application Server, you must install WebSphere Application Server. For detailed information on installing the application server, read about installing your application server environment.

About this task

Set up a web services development environment by completing the following actions.

Procedure

1. Set up the environment.

   Run the **setupCmdLine** script from the */profile_root/<application_server>/bin* directory.
2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:

```
set PATH=%WAS_PATH%;%PATH%
```

Results

You have set up an environment so that you can develop Web services.

What to do next

Implement web services applications. See the task overview for implementing web services applications information to learn about how to develop and implement a Java EE web service.

Generating Java artifacts for JAX-WS applications from a WSDL file

Use JAX-WS tools to generate the Java artifacts that are needed to develop JAX-WS web services when starting with a Web Services Description Language (WSDL) file.

Before you begin

When using a top-down development approach to developing Java API for XML-Based Web Services (JAX-WS) web services by starting with a Web Services Description Language (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

About this task

You can use the JAX-WS tool, **wsimport**, to process a WSDL file and generate portable Java artifacts that are used to create a web service. The portable Java artifacts created using the **wsimport** tool are:

- Service endpoint interface (SEI)
- Service class
- Exception class that is mapped from the `wsdl:fault` class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To

take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

In addition to using the tools from the command-line, you can invoke these JAX-WS tools from within the Ant build environments. Use the `com.sun.tools.ws.ant.WsImport` Ant task from within the Ant build environment to invoke the wsimport tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Procedure

Run the wsimport command to generate the portable client artifacts. The **wsimport** tool is located in the *app_server_root*\bin\ directory.
(Optional) Use the following options with the **wsimport** command:

- Use the **-verbose** option to see a list of generated files when you run the command.
- Use the **-keep** option to keep generated Java files.
- Use the **-wsdlLocation** option to specify the location of the WSDL file.

  Note: A best practice for ensuring that you produce a JAX-WS web services client enterprise archive (EAR) file that is portable to other systems is to package the WSDL document within the application module such as a web services client Java archive (JAR) file or a web application archive (WAR) file. You can specify a relative URI for the location of your WSDL file by using the `-wsdllocation` annotation attribute. For example, if your `MyService.wsdl` file is located in the `META-INF/wsdl/` directory, then run the **wsimport** tool and use the `-wsdllocation` option to specify the value to be used for the location of the WSDL file. This ensures that the generated artifacts contain the correct `-wsdllocation` information needed when the application is loaded into the administrative console; for example:

```
wsimport -keep -wsdllocation=META-INF/wsdl/MyService.wsdl
```

- Use the **-b** option if you are using WSDL or schema customizations to specify external binding files that contain your customizations.

  You can customize the bindings in your WSDL file to enable asynchronous mappings or attachments. To generate asynchronous interfaces, add the client-side only customization `enableAsyncMapping` binding declaration to the `wsdl:definitions` element or in an external binding file that is defined in the WSDL file. Use the `enableMIMEContent` binding declaration in your custom client or server binding file to enable or disable the default mime:content mapping rules. For additional information on custom binding declarations, see chapter 8 the JAX-WS specification.

Read about the **wsimport** command to learn more about this command and additional options that you can specify.

Results

You have the required Java artifacts to create a JAX-WS web service. To learn more about the usage, syntax, and parameters for the **wsimport** command, see the **wsimport** command for JAX-WS applications documentation.

Example

The following example illustrates how the **wsimport** command is used to process the sample Ping WSDL file to generate portable artifacts.

1. Copy the following `ping.wsdl` WSDL file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 * This program can be used, run, copied, modified and distributed
 * without royalty for the purpose of developing, using, marketing, or distributing.
 -->
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://com/ibm/was/wssample/sei/ping/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="PingService"
targetNamespace="http://com/ibm/was/wssample/sei/ping/">
<wsdl:types>
 <xsd:schema
  targetNamespace="http://com/ibm/was/wssample/sei/ping/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">


  <xsd:element name="pingStringInput">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="pingInput" type="xsd:string" />
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:schema>
</wsdl:types>
<wsdl:message name="pingOperationRequest">
 <wsdl:part element="tns:pingStringInput" name="parameter" />
</wsdl:message>
<wsdl:portType name="PingServicePortType">
 <wsdl:operation name="pingOperation">
  <wsdl:input message="tns:pingOperationRequest" />

 </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="PingSOAP" type="tns:PingServicePortType">
 <soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http" />
 <wsdl:operation name="pingOperation">
  <soap:operation soapAction="pingOperation" style="document" />
  <wsdl:input>
   <soap:body use="literal" />
  </wsdl:input>
 </wsdl:operation>
</wsdl:binding>
<wsdl:service name="PingService">
 <wsdl:port binding="tns:PingSOAP" name="PingServicePort">
  <soap:address
   location="http://localhost:9080/WSSampleSei/PingService" />
 </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

2. Run the **wsimport** command from the *app_server_root*\bin\ directory.

After generating the template files using the **wsimport** command, the following files are generated:

```
com\ibm\was\wssample\sei\ping\ObjectFactory.java
com\ibm\was\wssample\sei\ping\package-info.java
com\ibm\was\wssample\sei\ping\PingServicePortType.java
com\ibm\was\wssample\sei\ping\PingStringInput.java
com\ibm\was\wssample\sei\ping\PingService.java
```

The `ObjectFactory.java` file contains factory methods for each Java content interface and Java element interface generated in the associated ping package. The `package-info.java` file takes the targetNamespace value and creates the directory structure. The `PingServicePortType.java` file is the generated service endpoint interface (SEI) class that contains the `ping` method definition. The `PinStringInput.java` file contains the JAXB generated type values which are Java classes mapped from XML schema types. The `PingService.java` file is the generated service provider class file that is used by the JAX-WS client.

What to do next

Complete the implementation of your web service application by completing the JavaBeans or enterprise beans implementation.

wsimport command for JAX-WS applications

The wsimport command-line tool processes an existing Web Services Description Language (WSDL) file and generates the required artifacts for developing Java API for XML-Based Web Services (JAX-WS) web service applications. The generated artifacts are Java 5 compliant, making them portable across different Java versions and platforms.

The **wsimport** command-line tool supports the top-down approach to developing JAX-WS web services. When you start with an existing WSDL file, use the **wsimport** command-line tool to generate the required JAX-WS artifacts.

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

The **wsimport** tool reads an existing WSDL file and generates the following artifacts:

- Service Endpoint Interface (SEI) - The SEI is the annotated Java representation of the WSDL file for the web service. This interface is used for implementing JavaBeans endpoints or creating dynamic proxy client instances.
- `javax.xml.ws.Service` extension class - This is a generated class that extends the `javax.xml.ws.Service` class. This class is used to configure and create both dynamic proxy and dispatch instances.
- required data beans, including any Java Architecture for XML Binding (JAXB) beans that are required to model the web service data.

You can package the generated artifacts in a web application archive (WAR) file with the WSDL file and schema documents along with the endpoint implementation to be deployed.

Note: To correctly use the **wsimport** tool, you must adhere to the following requirements:

- You must define all your services within the main WSDL file. Services that are defined within an imported WSDL file are not processed by the **wsimport** tool.
- If you run the **wsimport** tool on a WSDL file that implements a Document or Literal style pattern, the complexTypes elements that define the input and output types must be composed of unique names to prevent naming conflicts in the parameter list for the operation..
- If you run the **wsimport** tool and pass a `?wsdl` Uniform Resource Identifier (URI) as a parameter for a WSDL file, ensure that you are using the actual resolved WSDL URI. The **wsimport** tool correctly resolves the `?wsdl` URI, but other relative URIs that are referenced might not resolve correctly.

In addition to using the tools from the command line, you can invoke these JAX-WS tools from within the Ant build environments. Use the `com.sun.tools.ws.ant.WsImport` Ant task from within the Ant build environment to invoke the wsimport tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Syntax

The command-line syntax is:

Parameters

The *WSDL_URI* is the only parameter that is required. The following parameters are optional for the **wsimport** command:

-b <path>
Specifies the external JAX-WS or JAXB binding files. You can specify multiple JAX-WS and JAXB binding files by using the `-b` option; however, each file must be specified with its own `-b` option.

-B <jaxbOption>
Specifies to pass this option to the JAXB schema compiler.

-catalog
Specifies the catalog file to resolve external entity references. It supports the TR9401, XCatalog, and the OASIS XML Catalog formats

-d <directory>
Specifies where to place the generated output files.

-extension
Specifies whether to accept custom extensions for functionality that are not specified by the JAX-WS specification. The use of custom extensions can result in applications that are not portable or do not interoperate with other implementations.

-help
Displays the help menu.

-httpproxy:<host>:<port>
Specifies an HTTP proxy. The default port value is 8080.

-keep
Specifies whether to keep the generated source files.

-p <package_name>
Specifies a target package with this command-line option and overrides any WSDL file and schema binding customization for the package name and the default package name algorithm defined in the JAX-WS specification.

-quiet
Specifies to suppress the **wsimport** output.

-s <directory>
Specifies the directory to place the generated source files.

-target <version>>
Specifies to generate code that is compliant with a specific JAX-WS specification level. Specify version `2.0` or `2.1` to generate code that is compliant with the JAX-WS 2.0 or JAX-WS 2.1 specification respectively. Specifying version `2.1` indicates to generate code that is compliant with the JAX-WS 2.1 specification. The default value is version `2.2` and generates compliant code for the JAXB 2.2 specification.

-verbose
Specifies to output messages about what the compiler is doing.

-version
Prints the version information. If you specify this option, only the version information is included in the output and normal command processing does not occur.

-wsdlLocation
Specifies the `@WebServiceClient.wsdlLocation` value.

Note: The **wsimport** tool does not set the `@WebService.wsdlLocation` value either by default or when the -wsdlLocation attribute is specified. The **wsimport** command-line tool updates the @WebServiceClient.wsdlLocation annotation only. You can manually update the @WebService.wsdlLocation annotation with a relative URL that specifies the location of the Web Services Description Language (WSDL) file. If the @WebService.wsdlLocation annotation is present on an endpoint implementation class, then the value must be a relative URL and the WSDL document that it references must be packaged with the application.

Note: If you specify an HTTPS URL for the -wsdlLocation parameter, the **wsimport** tool generates a service class with a no-argument constructor that is not valid. Avoid using the no-argument service constructor to instantiate your service. Instead, pass the HTTPS URL to one of the service class constructors that takes a WSDL URL for an argument; for example:

```
MyService("https://example.ibm.com/My?wsdl");
```

Enabling MTOM for JAX-WS web services

With Java API for XML-Based Web Services (JAX-WS), you can send binary attachments such as images or files along with web services requests. JAX-WS adds support for optimized transmission of binary data as specified by the SOAP Message Transmission Optimization Mechanism (MTOM) specification.

About this task

JAX-WS supports the use of SOAP Message Transmission Optimized Mechanism (MTOM) for sending binary attachment data. By enabling MTOM, you can send and receive binary data optimally without incurring the cost of data encoding needed to embed the binary data in an XML document.

The application server supports sending attachments using MTOM only for JAX-WS applications. This product also provides the ability to provide attachments with Web Services Security SOAP messages by using the new MTOM and XOP standards.

JAX-WS applications can send binary data as base64 or hexBinary encoded data contained within the XML document. However, to take advantage of the optimizations provided by MTOM, enable MTOM to send binary base64 data as attachments contained outside the XML document. MTOM optimization is not enabled by default. JAX-WS applications require separate configuration of both the client and the server artifacts to enable MTOM support.

Procedure

1. Develop Java artifacts for your JAX-WS application that includes an XML schema or Web Services Description Language (WSDL) file that represents your web services application data that includes a binary attachment.
   a. If you are starting with a WSDL file, develop Java artifacts from a WSDL file by using the wsimport command to generate the required JAX-WS portable artifacts.
   b. If you are starting with JavaBeans components, develop Java artifacts for JAX-WS applications and optionally generate a WSDL file using the wsgen command. The XML schema or WSDL file includes a `xsd:base64Binary` or `xsd:hexBinary` element definition for the binary data.
   c. You can also include the `xmime:expectedContentTypes` attribute on the element to affect the mapping by JAXB.
2. Enable MTOM on your endpoint implementation class using one of the following methods:
   - Use the @MTOM annotation on the endpoint.

     To enable MTOM on an endpoint, use the @MTOM (javax.xml.ws.soap.MTOM) annotation on the endpoint. The @MTOM annotation has two parameters, `enabled` and `threshold`. The `enabled` parameter has a boolean value and indicates if MTOM is enabled for the JAX-WS endpoint. The

threshold parameter has an integer value, that must be greater than or equal to zero. When MTOM is enabled, any binary data whose size, in bytes, exceeds the threshold value is XML-binary Optimized Packaging (XOP) encoded or sent as an attachment. When the message size is less than the threshold value, the message is inlined in the XML document as either base64 or hexBinary data.

The following example snippet illustrates adding the @MTOM annotation so that MTOM is enabled for the JAX-WS MyServiceImpl endpoint and specifies a threshold value of 2048 bytes:

```
@MTOM(enabled=true, threshold=2048)
@WebService
public class MyServiceImpl {
...
}
```

Additionally, you can use the @BindingType (javax.xml.ws.BindingType) annotation on an endpoint implementation class to specify that the endpoint supports one of the MTOM binding types so that the response messages are MTOM-enabled. The javax.xml.ws.SOAPBinding class defines two different constants, SOAP11HTTP_MTOM_BINDING and SOAP12HTTP_MTOM_BINDING that you can use for the value of the @BindingType annotation; for example:

```
// This example is for SOAP version 1.1.
@BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING)
@WebService
public class MyServiceImpl {
...
}

// This example is for SOAP version 1.2.
@BindingType(value = SOAPBinding.SOAP12HTTP_MTOM_BINDING)
@WebService
public class MyServiceImpl {
...
}
```

The presence and value of an @MTOM annotation overrides the value of the @BindingType annotation. For example, if the @BindingType indicates MTOM is enabled, but an @MTOM annotation is present with an enabled value of false, then MTOM is not enabled.

- Use the <enable-mtom> and <mtom-threshold> deployment descriptor elements.

  You can use the <enable-mtom> and <mtom-threshold> elements within the <port-component> element in the webservices.xml deployment descriptor as an alternative to using the @MTOM annotation on the service endpoint implementation class; for example:

```
<port-component>
   <port-component-name>MyPort1</port-component-name>
   <enable-mtom>true</enable-mtom>
   <mtom-threshold>2048</mtom-threshold>
   <service-impl-bean>
    <servlet-link>MyPort1ImplBean</servlet-link>
   </service-impl-bean>
</port-component>
```

Note: The deployment descriptor elements take precedence over the corresponding attributes in the MTOM annotation. For example, if the enabled attribute is set to true in the annotation, but the <enable-mtom> element is set to false in the webservices.xml file, MTOM is not enabled for the corresponding endpoint.

3. Enable MTOM on your client to optimize the binary messages that are sent from the client to the server. Use one of the following methods to enable MTOM on your client:

   - Enable MTOM on a Dispatch client.

     The following examples use SOAP version 1.1.

     The first method uses SOAPBinding.setMTOMEnabled(); for example:

```
SOAPBinding binding = (SOAPBinding)dispatch.getBinding();
    binding.setMTOMEnabled(true);
```

     The second method uses Service.addPort; for example:

```
    Service svc = Service.create(serviceName);
    svc.addPort(portName,SOAPBinding.SOAP11HTTP_MTOM_BINDING,endpointUrl);
```

     The third method uses MTOMFeature; for example:

```
   MTOMFeature mtom = new MTOMFeature(true, 2048);
    Service svc = Service.create(serviceName);
    svc.addPort(portName, SOAPBinding.SOAP11_HTTP_BINDING, endpointUrl);
    Dispatch<Source> dsp = svc.createDispatch(portName, Source.class, Service.Mode.PAYLOAD, mtom);
```

- Enable MTOM on a Dynamic Proxy client.

```
// Create a BindingProvider bp from a proxy port.
  Service svc = Service.create(serviceName);
  MtomSample proxy = svc.getPort(portName, MtomSample.class);
  BindingProvider bp = (BindingProvider) proxy;

  //Enable MTOM using the SOAPBinding.
  MtomSample proxy = svc.getPort(portName, MtomSample.class);
  BindingProvider bp = (BindingProvider) proxy;
  SOAPBinding binding = (SOAPBinding) bp.getBinding();
  binding.setMTOMEnabled(true);

  //Or, you can enable MTOM with the MTOMFeature.
  MTOMFeature mtom = new MTOMFeature();
  MtomSample proxy = svc.getPort(portName, MtomSample.class, mtom);
```

- Enable MTOM on your client using the @MTOM annotation; for example:

```
public class MyClientApplication {

    // Enable MTOM for a port-comonent-ref resource injection.
     @MTOM(enabled=true, threshold=1024)
     @WebServiceRef(MyService.class)
     private MyPortType myPort;
    ...
}
```

- Enable MTOM on your client using deployment descriptor elements within a port-component-ref element; for example:

```
<service-ref>
    <service-ref-name>service/MyPortComponentRef</service-ref-name>
    <service-interface>com.example.MyService</service-ref-interface>
    <port-component-ref>
        <service-endpoint-interface>com.example.MyPortType</service-endpoint-interface>
        <enable-mtom>true</enable-mtom>
        <mtom-threshold>1024</mtom-threshold>
    </port-component-ref>
</service-ref>
```

Note: The deployment descriptor elements take precedence over the corresponding attributes in the MTOM annotation. For example, if the `enabled` attribute is set to `true` in the annotation, but the `<enable-mtom>` element is set to `false` in the deployment descriptor entry for the service-ref of the client, MTOM is not enabled for that service reference.

Results

You have developed a JAX-WS web services server and client application that optimally sends and receives binary data using MTOM.

Example

The following example illustrates enabling MTOM support on both the web services client and server endpoint when starting with an WSDL file.

1. Locate the WSDL file containing an `xsd:base64Binary` element. The following example is a portion of a WSDL file that contains an `xsd:base64Binary` element.

```
  <types>
    ........
    <xs:complexType name="ImageDepot">
        <xs:sequence>
            <xs:element name="imageData"
                        type="xs:base64Binary"
                        xmime:expectedContentTypes="image/jpeg"/>

        </xs:sequence>
    </xs:complexType>
    ........
</types>
```

2. Run the **wsimport** command from the *app_server_root*\bin\ directory against the WSDL file to generate a set of JAX-WS portable artifacts.

   Depending on the `expectedContentTypes` value contained in the WSDL file, the JAXB artifacts generated are in the Java type as described in the following table:

Table 130. Mapping of MIME type and Java type. Describes the mapping between MIME types and Java types.

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
/	javax.activation.DataHandler

3. Use the JAXB artifacts in the same manner as in any other JAX-WS application. Use these beans to send binary data from both the Dispatch and the Dynamic Proxy client APIs.
4. Enable MTOM on a Dispatch client.

```
//Create the Dispatch instance.
        JAXBContext jbc = JAXBContext.newInstance("org.apache.axis2.jaxws.sample.mtom");
        Dispatch<Object> dispatch = svc.createDispatch(portName, jbc, Service.Mode.PAYLOAD);

//Enable MTOM.
        SOAPBinding binding = (SOAPBinding) dispatch.getBinding();
        binding.setMTOMEnabled(true);
```

5. Enable MTOM on a Dynamic Proxy client.

```
//Create the Dynamic Proxy instance.
        Service svc = Service.create(serviceName);
        MtomSample proxy = svc.getPort(portName, MtomSample.class);

//Enable MTOM.
        BindingProvider bp = (BindingProvider) proxy;
        SOAPBinding binding = (SOAPBinding) bp.getBinding();
        binding.setMTOMEnabled(true);
```

   Now that you have enabled the JAX-WS client for MTOM, messages sent to the server have MTOM enabled. However, for the server to respond back to the client using MTOM, you must enable MTOM on the endpoint.

6. Enable MTOM on the endpoint implementation class.

```
@MTOM(enabled, threshold=4096)
@WebService (endpointInterface="org.apache.axis2.jaxws.sample.mtom.MtomSample")

public class MtomSampleService implements MtomSample {
   ....
}
```

   The `jaxax.xml.ws.SOAPBinding` class has a static member for each of the supported binding types. Include either the `SOAP11HTTP_MTOM_BINDING` or the `SOAP12HTTP_MTOM_BINDING` as the value for the `@BindingType` annotation. This value enables all server responses to have MTOM enabled.

When you enable MTOM on the server and the client, the binary data that represents the attachment is included as a Multipurpose Internet Mail Extensions (MIME) attachment to the SOAP message. Without MTOM, the same data is encoded in the format that describes the XML schema, either base64 or hexadecimal encoding, and included inline in the XML document.

This example illustrates an MTOM enabled SOAP version 1.1 message with an attachment. The `type` and `content-type` attributes both have the value, `application/xop+xml`, which indicates that the message was successfully optimized using XML-binary Optimized packaging (XOP) when MTOM was enabled. This example demonstrates how the optimized message looks on the wire with MTOM enabled.

```
... other transport headers ...
Content-Type: multipart/related; boundary=MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812;
type="application/xop+xml"; start="
<0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>"; start-info="text/xml"; charset=UTF-8

--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: application/xop+xml; charset=UTF-8; type="text/xml";
content-transfer-encoding: binary
content-id:
   <0.urn:uuid:0FE43E4D025F0BF3DC11582467646813@apache.org>

<?xml version="1.0" encoding="UTF-8"?>
         <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
```

```
            <soapenv:Header/>
            <soapenv:Body>
               <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
                  <input>
                     <imageData>
                        <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:1.urn:uuid:0FE43E4D025F0BF3DC11582467646811@apache.org"/>
                     </imageData>
                  </input>
               </sendImage>
            </soapenv:Body>
         </soapenv:Envelope>
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812
content-type: text/plain
content-transfer-encoding: binary
content-id:
         <1.urn:uuid:0FE43E4D025F0BF3DC11582467646811@apache.org>

... binary data goes here ...
--MIMEBoundaryurn_uuid_0FE43E4D025F0BF3DC11582467646812--
```

In contrast, this example demonstrates a SOAP version 1.1 message on the wire without MTOM enabled. The binary data is included in the body of the SOAP message, and the SOAP message is not optimized.

```
... other transport headers ...
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Header/>
      <soapenv:Body>
         <sendImage xmlns="http://org/apache/axis2/jaxws/sample/mtom">
            <input>
               <imageData>R0lGAD1 ... more base64 encoded data ... KTJk8giAAA7</imageData>
            </input>
         </sendImage>
      </soapenv:Body>
   </soapenv:Envelope>
```

For additional information, refer to the Samples section of the Information Center which includes a sample that demonstrates the use of MTOM with JAX-WS web services.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root

Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root

Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root

Refers to the installation root directory for Web Server Plug-ins.

profile_root

Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root

Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Enforcing adherence to WSDL bindings in JAX-WS web services

Java API for XML-Based Web Services (JAX-WS) Version 2.1 introduced the concept of *features* as a way to programmatically control specific functions and behaviors. The RespectBindingFeature is one of the supported standard features. You can use the RespectBindingFeature to control whether a JAX-WS implementation is required to respect the contents of a Web Services Description Language (WSDL) binding that is associated with an endpoint.

About this task

While WSDL documents are often used during the development process, the actual enforcement of the use of the WSDL document specifications, when they are provided, at run time has not been well defined in versions of the JAX-WS specification previous to Version 2.1. The JAX-WS Version 2.1 specification added the feature, RespectBindingFeature, to clarify the impact of the wsdl:binding in a JAX-WS runtime environment.

Enabling the feature, RespectBindingFeature, causes the JAX-WS runtime environment to inspect the wsdl:binding for an endpoint at run time to ensure that the binding of service endpoint interface (SEI) parameters and return values is respected. Additionally, this feature ensures that all required wsdl:binding extensions are either understood and used by the runtime environment, or the extensions have been explicitly disabled by the application. Your JAX-WS application can disable a specific wsdl:binding extension that has a defined WebServiceFeature interface by using the appropriate annotation that is associated with that feature, using an API that accepts the javax.xml.ws.WebServiceFeature interface, or configuring the deployment descriptors.

When the RespectBindingFeature feature is not enabled, which is the default, the runtime environment can choose whether any part of the wsdl:binding is enforced.

Procedure

1. Develop Java artifacts for your JAX-WS application that includes a Web Services Description Language (WSDL) file that represents your web services application.
   a. If you are starting with a WSDL file, develop Java artifacts from a WSDL file by using the wsimport command to generate the required JAX-WS portable artifacts.
   b. If you are starting with JavaBeans components, develop Java artifacts for JAX-WS applications and generate a WSDL file using the wsgen command.
2. If you want to enable RespectBindingFeature on your endpoint implementation class, use one of the following methods:
   - Use the @RespectBinding annotation on the endpoint.

     To enable RespectBinding on an endpoint, use the @RespectBinding (javax.xml.ws.RespectBinding) annotation on the endpoint. The @RespectBinding annotation has only one parameter, enabled. This parameter is optional. The enabled parameter has a Boolean value and indicates if RespectBindingFeature is enabled for the JAX-WS endpoint.

     The following example snippet illustrates adding the @RespectBinding annotation for the JAX-WS MyServiceImpl endpoint:

```
@RespectBinding(enabled=true))
@WebService
public class MyServiceImpl {
...
}
```

- Use the <respect-binding> deployment descriptor element.

  You can use the <respect-binding> element within the <port-component> element in the webservices.xml deployment descriptor as an alternative to using the @RespectBinding annotation on the service endpoint implementation class; for example:

```
<port-component>
     <port-component-name>MyPort1</port-component-name>
     <respect-binding>
         <enabled>true</enabled>
     </respect-binding>
     <service-impl-bean>
         <servlet-link>MyPort1ImplBean</servlet-link>
     </service-impl-bean>
</port-component>
```

3. If you want to enable RespectBindingFeature on your client, use one of the following methods:
   - Enable RespectBindingFeature on a dispatch client; for example:

```
RespectBindingFeature respectBinding = new RespectBindingFeature();
Service svc = Service.create(serviceName);
svc.addPort(portName, SOAPBinding.SOAP11_HTTP_BINDING, endpointUrl);
Dispatch<Source> dsp = svc.createDispatch(portName, Source.class, Service.Mode.PAYLOAD, respectBinding);
```

   - Enable RespectBindingFeature on a dynamic proxy client; for example:

```
// Create a dynamic proxy with RespectBinding enabled.
  Service svc = Service.create(serviceName);
  RespectBindingFeature respectBinding = new RespectBindingFeature();
  RespectBindingSample proxy = svc.getPort(portName, RespectBindingSample.class, respectBinding);
```

   - Enable RespectBindingFeature on your client using the @RespectBinding annotation; for example:

```
public class MyClientApplication {

               // Enable RespectBinding for a port-component-ref resource injection.
                   @RespectBinding(enabled=true)
                   @WebServiceRef(MyService.class)
                   private MyPortType myPort;
               ...
}
```

   - Enable RespectBindingFeature on your client using deployment descriptor elements within a port-component-ref element; for example:

```
<service-ref>
    <service-ref-name>service/MyPortComponentRef</service-ref-name>
    <service-interface>com.example.MyService</service-ref-interface>
    <port-component-ref>
        <service-endpoint-interface>com.example.MyPortType</service-endpoint-interface>
        <respect-binding>
            <enabled>true</enabled>
        </respect-binding>
    </port-component-ref>
</service-ref>
```

Results

By implementing the feature, RespectBindingFeature, you have specified to enforce adherence of the contents of a WSDL binding that is associated with an endpoint for your JAX-WS application.

Developing a webservices.xml deployment descriptor for JAX-WS applications

Deployment descriptors are standard text files, formatted using XML and packaged in a web services application. You can optionally use the webservices.xml deployment descriptor to augment or override application metadata specified in annotations within Java API for XML-Based Web Services (JAX-WS) web services.

About this task

Similar to Java API for XML-based RPC (JAX-RPC) Web services, you can use deployment descriptors to describe JAX-WS web services. For JAX-WS web services, the use of the webservices.xml deployment

descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

A JAX-WS web service requires that you annotate your Java class with the `javax.jws.WebService` annotation or the `javax.jws.WebServiceProvider` annotation for Provider endpoints. You can use server-side deployment descriptors to override corresponding attributes of the annotation or to enhance information in annotations. There is a defined relationship between the deployment descriptor elements and the @WebService and @WebServiceProvider annotations. Refer to section 5.3 in the Web Services for Java Platform, Enterprise Edition (Java EE) specification, Version 1.2 for detailed information regarding the deployment descriptor elements and the mapping to the @WebService and @WebServiceProvider annotation attributes. There are also elements in the `webservice.xml` deployment descriptor that map to other annotations. For example, the deployment descriptor element <protocol-binding> maps to the @BindingType annotation, and the deployment descriptor element <enable-mtom> maps to the @MTOM annotation. For more information regarding the web services deployment descriptor elements, see section 7.1 in the Web Services for Java Platform, Enterprise Edition (Java EE) specification.

Procedure

Use assembly tools to generate the `webservice.xml` deployment descriptor.

Results

You have deployment descriptor templates that you can use to override JAX-WS annotation attributes or specify attributes that are not defined by the annotation.

Example

In the following example, the service implementation class for a JAX-WS web service includes the @WebService annotation:

```
@WebService(wsdlLocation="http://myhost.com/location/of/the/wsdl/ExampleService.wsdl")
```

The associated `webservices.xml` deployment descriptor specifies a different filename for the WSDL document as follows:

```
<webservices>
<webservice-description>
<webservice-description-name>ExampleService</webservice-description-name>
<wsdl-file>META-INF/wsdl/ExampleService.wsdl</wsdl-file>
...
</webservice-description>
</webservices>
```

The value that is specified in the deployment descriptor, `META-INF/wsdl/ExampleService.wsdl`, overrides the annotation value.

What to do next

Configure the `webservice.xml` deployment descriptor. After you configure the deployment descriptors, you must assemble the Web services application for deployment.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
Refers to the installation root directory for Web Server Plug-ins.

profile_root
Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Completing the JavaBeans implementation for JAX-WS applications

After you have developed the Java artifacts necessary to develop a Java API for XML-Based Web Services (JAX-WS) web service, you must complete the JavaBeans implementation to assemble a web application archive (WAR) file. The resulting WAR file contains the JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Generate Java artifacts for JAX-WS applications and optionally generate a WSDL file using the **wsgen** command-line tool. You can also optionally use deployment descriptors to augment or override binding information contained in annotations for JAX-WS web services.

About this task

For JAX-WS applications, complete the JavaBeans implementation by writing your business application.

Procedure

1. Write the JavaBeans implementation. The JavaBeans implementation is not generated by JAX-WS tooling.
2. Compile all the Java classes.

Results

You have now written your JavaBeans implementation to complete your web service application.

What to do next

After completing the JavaBeans implementation, assemble your web services application.

Completing the EJB implementation for JAX-WS applications

After you have developed the Java artifacts necessary to develop a Java API for XML-Based Web Services (JAX-WS) web service, you must complete the Enterprise JavaBeans (EJB) implementation to assemble a Java archive (JAR) file. The resulting JAR file contains the Enterprise JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Generate Java artifacts for JAX-WS applications and optionally generate a WSDL file using the **wsgen** command-line tool. You can also optionally use deployment descriptors to augment or override binding information contained in annotations for JAX-WS web services.

About this task

For JAX-WS applications, complete the enterprise beans implementation by writing your business application.

Procedure

1. Write the enterprise beans implementation. The enterprise beans implementation is not generated by JAX-WS tooling.
2. Compile all the Java classes.

Results

You have now written your enterprise beans implementation to complete your web service application.

What to do next

After completing the enterprise beans implementation, assemble your web services application.

Developing JAX-WS clients

Developing a JAX-WS client from a WSDL file

Java API for XML-Based Web Services (JAX-WS) tooling supports generating Java artifacts you need to develop static JAX-WS web services clients when starting with a Web Services Description Language (WSDL) file.

Before you begin

When you use a top-down development approach to developing JAX-WS web services by starting with a WSDL file, you must obtain the Uniform Resource Locator (URL) for the WSDL file.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

About this task

The static client programming model for JAX-WS is the called the dynamic proxy client. The dynamic proxy client invokes a web service based on a service endpoint interface that is provided. After you create

the proxy, the client application can invoke methods on the proxy just like a standard implementation of those interfaces. For JAX-WS web service clients using the dynamic proxy programming model, use the JAX-WS tool, **wsimport**, to process a WSDL file and generate portable Java artifacts that are used to create a web service client. Create the following portable Java artifacts using the **wsimport** tool:

- Service endpoint interface (SEI)
- Service class
- Exception class that is mapped from the wsdl:fault class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

Note: The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere Application Server provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The **wsimport**, **wsgen**, **schemagen** and **xjc** command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. On some occasions, the artifacts generated by both the tooling provided by WebSphere Application Server and the JDK support the same levels of the specifications. In general, the artifacts generated by the JDK tools are portable across other compliant runtime environments. However, it is a best practice to use the tools provided with this product to achieve seamless integration within the WebSphere Application Server environment and to take advantage of the features that may be only supported in WebSphere Application Server. To take advantage of JAX-WS and JAXB V2.2 tooling, use the tools provided with the application server that are located in the *app_server_root*\bin\ directory.

In addition to using the tools from the command-line, you can invoke these JAX-WS tools from within the Ant build environments. Use the `com.sun.tools.ws.ant.WsImport` Ant task from within the Ant build environment to invoke the wsimport tool. To function properly, this Ant task requires that you invoke Ant using the ws_ant script.

Procedure

Run the wsimport command to generate the portable client artifacts. The **wsimport** tool is located in the *app_server_root*\bin\ directory.
(Optional) Use the following options with the **wsimport** command:

- Use the **-verbose** option to see a list of generated files when you run the command.
- Use the **-keep** option to keep generated Java files.
- Use the **-wsdlLocation** option to specify the location of the WSDL file.

  Note: A best practice for ensuring that you produce a JAX-WS web services client enterprise archive (EAR) file that is portable to other systems is to package the WSDL document within the application module such as a web services client Java archive (JAR) file or a web application archive (WAR) file. You can specify a relative URI for the location of your WSDL file by using the `-wsdllocation` annotation attribute. For example, if your `MyService.wsdl` file is located in the `META-INF/wsdl/ directory`, then run the **wsimport** tool and use the `-wsdllocation` option to specify the value to be used for the location of the WSDL file. This ensures that the generated artifacts contain the correct `-wsdllocation` information needed when the application is loaded into the administrative console; for example:

```
wsimport -keep -wsdllocation /META-INF/wsdl/MyService.wsdl
```

- Use the **-b** option if you are using WSDL or schema customizations to specify external binding files that contain your customizations.

You can customize the bindings in your WSDL file to enable asynchronous mappings or attachment files. To generate asynchronous interfaces, add the client-side only customization `enableAsyncMapping` binding declaration to the `wsdl:definitions` element or in an external binding file that is defined in the WSDL file. Use the `enableMIMEContent` binding declaration in your custom client or server binding file to enable or disable the default mime:content mapping rules. For additional information on custom binding declarations, see chapter 8 the JAX-WS specification.

Read about the **wsimport** command to learn more about this command and additional options that you can specify.

Results

You have the generated Java artifacts to create a JAX-WS client that can invoke JAX-WS web services. To learn more about the usage, syntax, and parameters for the **wsimport** command, see the **wsimport** command for JAX-WS applications documentation.

Example

The following example illustrates how the **wsimport** command is used to process the sample `ping.wsdl` file to generate portable artifacts.

1. Copy the following `ping.wsdl` file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 * This program can be used, run, copied, modified and distributed
 * without royalty for the purpose of developing, using, marketing, or distributing.
 -->
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://com/ibm/was/wssample/sei/ping/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="PingService"
 targetNamespace="http://com/ibm/was/wssample/sei/ping/">
 <wsdl:types>
  <xsd:schema
   targetNamespace="http://com/ibm/was/wssample/sei/ping/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">


   <xsd:element name="pingStringInput">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="pingInput" type="xsd:string" />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>
 </wsdl:types>
 <wsdl:message name="pingOperationRequest">
  <wsdl:part element="tns:pingStringInput" name="parameter" />
 </wsdl:message>
 <wsdl:portType name="PingServicePortType">
  <wsdl:operation name="pingOperation">
   <wsdl:input message="tns:pingOperationRequest" />

  </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="PingSOAP" type="tns:PingServicePortType">
  <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="pingOperation">
   <soap:operation soapAction="pingOperation" style="document" />
   <wsdl:input>
    <soap:body use="literal" />
   </wsdl:input>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="PingService">
  <wsdl:port binding="tns:PingSOAP" name="PingServicePort">
   <soap:address
```

```
    location="http://localhost:9080/WSSampleSei/PingService" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

2. Run the **wsimport** command from the *app_server_root*\bin\ directory.

After generating the template files from the **wsimport** command, the following files are generated:

```
com\ibm\was\wssample\sei\ping\ObjectFactory.java
com\ibm\was\wssample\sei\ping\package-info.java
com\ibm\was\wssample\sei\ping\PingServicePortType.java
com\ibm\was\wssample\sei\ping\PingService.java
com\ibm\was\wssample\sei\ping\PingStringInput.java
```

The `ObjectFactory.java`, `PingService.java`, and `PingServicePortType.java` files are the generated Java class files to use when you package the Java artifacts with your client implementation inside a Java archive (JAR) or a web application archive (WAR) file.

What to do next

Complete the client implementation.

Developing deployment descriptors for a JAX-WS client

Deployment descriptors are standard text files, formatted using XML and packaged in a web services application. You can optionally use the Web Services for Java Platform, Enterprise Edition (Java EE) specification (JSR 109) service reference deployment descriptor to augment or override application metadata specified in annotations within Java API for XML-Based Web Services (JAX-WS) web services client.

Before you begin

You must first generate the web services client artifacts from a Web Services Description Language (WSDL) file using the **wsimport** command.

About this task

You can add `service-ref` entries within the `application-client.xml`, `web.xml`, or `ejb-jar.xml` Java EE deployment descriptors. A `service-ref` entry represents a reference to a web service that is used by a Java EE component in Web, Enterprise JavaBeans (EJB) or application client containers. A `service-ref` entry has a JNDI name that is used to lookup the service. Specifying the `service-ref` entry enables the client applications to locate the service using a JNDI lookup and you can also use these service references for resource injection.

For each `service-ref` entry found in one of the deployment descriptors, the corresponding service object is bound into the JNDI namespace and the port information is included, if specified. The JAX-WS client can now perform a JNDI lookup to retrieve either a JAX-WS service or port instance.

When defining a `service-ref` that represents a JAX-WS service, use the `javax.xml.ws.Service` subclass that is generated by the **wsimport** tool as the `service-interface` value. This is the class that contains the @WebServiceClient annotation. When defining a `service-ref` that represents a JAX-WS port, the `service-interface` value is still the javax.xml.ws.Service subclass generated by the **wsimport** tool, and the `service-ref-type` value specifies the service endpoint interface (SEI) class used by the port. The SEI class is also generated by **wsimport**, and it is annotated with the @WebService annotation.

Procedure

1. Define the `service-ref` entry in your `application-client.xml`, `web.xml`, or `ejb-jar.xml` deployment descriptors.

For example, suppose a web application archive (WAR) file contains a `WEB-INF/web.xml` deployment descriptor that includes the following `service-ref` entries:

```
<service-ref>
  <service-ref-name>service/ExampleService</service-ref-name>
  <service-interface>com.ibm.sample.ExampleService</service-interface>
</service-ref>

<service-ref>
  <service-ref-name>service/ExamplePort</service-ref-name>
  <service-interface>com.ibm.sample.ExampleService</service-interface>
  <service-ref-type>com.ibm.sample.ExamplePort</service-ref-type>
</service-ref>

<service-ref>
  <service-ref-name>service/ExamplePortInjected</service-ref-name>
  <service-interface>com.ibm.sample.ExampleService</service-interface>
  <service-ref-type>com.ibm.sample.ExamplePort</service-ref-type>

  <injection-target>
    <injection-target-class>com.ibm.sample.J2EEClient</injection-target-class>
    <injection-target-name>injectedPort</injection-target-name>
  </injection-target>
</service-ref>
```

In this example, `com.ibm.sample.ExampleService` is a generated JAX-WS service class and this class must be a subclass of javax.xml.ws.Service. Additionally, the `ExampleService.getPort()` method returns an instance of `com.ibm.sample.ExamplePort`.

2. Use the deployment descriptors within your web services client application to customize your application. The following code fragments are examples of how your client application can use the `service-ref` entries within the WAR module:

```
import javax.xml.ws.Service;
import com.ibm.sample.ExampleService;
import com.ibm.sample.ExamplePort;

// Create an InitialContext object for doing JNDI lookups.
InitialContext ic = new InitialContext();

// Client obtains an instance of the generic service class using JNDI.
Service genericService =
(Service) ic.lookup("java:comp/env/service/ExampleService");

// Client obtains an instance of the generated service class using JNDI.
ExampleService exampleService =
(ExampleService) ic.lookup("java:comp/env/service/ExampleService");

// Client obtains an instance of the port using JNDI.
ExamplePort ExamplePort =
(ExamplePort) ic.lookup("java:comp/env/service/ExamplePort");

// The container injects an instance of ExamplePort based on the client deployment descriptor
private ExamplePort injectedPort;
```

Results

You can now use the service references that were defined in the deployment descriptor within your client application. Additionally, you can use deployment descriptors to augment or override information specified by `@WebServiceRef` or `@Resource` annotations.

The `<lookup-name>` deployment descriptor element is new in Java EE 6, and is used to indirectly refer to an already-defined service reference. When the `<lookup-name>` element is used, only the `<service-ref-name>` element may also be specified, and no other child elements of `<service-ref>` may be defined.

The following example shows a `service-ref` entry within a `WEB-INF/web.xml` file which defines a reference to a JAX-WS service, as well as a `service-ref` entry within the same `web.xml` file which defines an indirect reference to the first `service-ref`:

```
<service-ref>
  <service-ref-name>service/ExampleService</service-ref-name>
  <service-interface>com.ibm.sample.ExampleService</service-interface>
  <service-ref-type>com.ibm.sample.ExampleServicePortType</service-ref-type>
  <wsdl-file>WEB-INF/wsdl/ExampleService.wsdl</wsdl-file>
</service-ref>
```

```
<service-ref>
  <service-ref-name>service/ExampleService2</service-ref>
  <lookup-name>java:comp/env/service/ExampleService</lookup-name>
</service-ref>
```

Assuming the above `service-ref`s are defined in the `WEB-INF/web.xml` file, the client application could perform a JNDI lookup using the name `java:comp/env/service/ExampleService2`, and the result would be a reference to the `ExampleService` service defined in the WSDL document `WEB-INF/wsdl/ExampleService.wsdl`, as defined in the first `service-ref`.

What to do next

Complete the client implementation by writing your client application code that is used to invoke the web service.

Developing a dynamic client using JAX-WS APIs

Java API for XML-Based Web Services (JAX-WS) provides support for the dynamic invocation of service endpoint operations.

About this task

JAX-WS provides a new dynamic Dispatch client API that is more generic and offers more flexibility than the existing Java API for XML-based RPC (JAX-RPC)-based Dynamic Invocation Interface (DII). The Dispatch client interface, javax.xml.ws.Dispatch, is an XML messaging oriented client that is intended for advanced XML developers who prefer to work at the XML level using XML constructs. To write a Dispatch client, you must have expertise with the Dispatch client APIs, the supported object types, and knowledge of the message representations for the associated Web Services Description Language (WSDL) file.

The Dispatch API can send data in either `PAYLOAD` or `MESSAGE` mode. When using the `PAYLOAD` mode, the Dispatch client is only responsible for providing the contents of the <soap:Body> and JAX-WS includes the input payload in a <soap:Envelope> element. When using the `MESSAGE` mode, the Dispatch client is responsible for providing the entire SOAP envelope.

The Dispatch client API requires application clients to construct messages or payloads as XML and requires a detailed knowledge of the message or message payload. The Dispatch client can use HTTP bindings when using Source objects, Java Architecture for XML Binding (JAXB) objects, or data source objects. The Dispatch client supports the following types of objects:

- javax.xml.transform.Source: Use Source objects to enable clients to use XML APIs directly. You can use Source objects with SOAP and HTTP bindings.
- JAXB objects: Use JAXB objects so that clients can use JAXB objects that are generated from an XML schema to create and manipulate XML with JAX-WS applications. JAXB objects can only be used with SOAP and HTTP bindings.
- javax.xml.soap.SOAPMessage: Use SOAPMessage objects so that clients can work with SOAP messages. You can only use SOAPMessage objects with SOAP version 1.1 or SOAP version 1.2 bindings.
- javax.activation.DataSource: Use DataSource objects so that clients can work with Multipurpose Internet Mail Extension (MIME) messages. Use DataSource only with HTTP bindings.

The Dispatch API uses the concept of generics that are introduced in Java SE Runtime Environment (JRE) 6. For each of the `invoke()` methods on the Dispatch interface, generics are used to determine the return type.

Procedure

1. Determine if you want your dynamic client to send data in `PAYLOAD` or `MESSAGE` mode.
2. Create a service instance and add at least one port to it. The port carries the protocol binding and service endpoint address information.

3. Create a Dispatch<T> object using either the Service.Mode.PAYLOAD method or the Service.Mode.MESSAGE method.
4. Configure the request context properties on the javax.xml.ws.BindingProvider interface. Use the request context to specify additional properties such as enabling HTTP authentication or specifying the endpoint address.
5. Optional: Set the jaxws.response.throwExceptionIfSOAPFault property on the RequestContext option for the Dispatch API to `Boolean.TRUE` if you do not want the JAX-WS runtime to throw a SOAPFaultException.

   The following example illustrates how to specify this property on the RequestContext option for the Dispatch API

```
Dispatch<OMElement> dispatch = service.createDispatch(
portName, OMElement.class, Mode.MESSAGE);
BindingProvider bp = (BindingProvider)dispatch;
bp.getRequestContext().put(
"jaxws.response.throwExceptionIfSOAPFault", Boolean.FALSE);
```

6. Compose the client request message for the dynamic client.
7. Invoke the service endpoint with the Dispatch client either synchronously or asynchronously.
8. Process the response message from the service.

Results

You have developed a dynamic JAX-WS client using the Dispatch API. Refer to Chapter 4, section 3 of the JAX-WS 2.0 specification for more information on using a Dispatch client.

JAX-WS dynamic ports, which are those added using the service method addPort, might have additional memory requirements starting in WebSphere Application Server Version 8.0. In previous releases, a single instance of a dynamic port could be shared across multiple service instances. In version 8.x, dynamic ports are now scoped to the instance of the service that added them. If a JAX-WS client has multiple service instances which refer to a dynamic port of the same name, those instances are no longer shared. This can potentially increase the memory requirements for that client. The memory used by dynamic ports is released when the service instance goes out of scope. However, if you encounter issues related to increased memory usage, it is possible to revert the behavior so that dynamic ports are again shared across service instances. To do this, set the system property `jaxws.share.dynamic.ports.enable` to the value `true`. However, note that doing so can cause some other issues, such as having policy set attachments incorrectly applied across shared dynamic ports. If you set this flag to `true` and encounter some of these issues, then you should remove the flag setting. In previous releases, if a SOAP Action was not provided by the Dispatch client application, the correct SOAP Action was not sent on the outbound message. Instead, the SOAP action was set to an anonymous operation. Starting in WebSphere Application Server Version 8, if the SOAP Action is not provided by the Dispatch client application, the JAX-WS runtime environment parses the outgoing message. It determines the operation being invoked and uses that information to determine the appropriate value for the SOAP Action. The operation resolution of the outbound message is based on the SOAP Body and the message encoding, for example, Doc/Lit/Bare, Doc/Lit/Wrapped. Since this parsing can be expensive, a property can be set. To always disable the parsing, set the property at the System level. To disable parsing on a per-message basis, set the property on the JAX-WS Request Message Context. The property is defined as a constant `org.apache.axis2.jaxws.Constants.DISPATCH_CLIENT_OUTBOUND_RESOLUTION` with a String value of `jaxws.dispatch.outbound.operation.resolution.enable`. The default value of the property is null which is interpreted as the String `true`, which enables the outbound operation resolution. Setting the property to `false` disables the outbound operation resolution. If parsing is disabled, then the SOAP Action in the outbound message is set to an anonymous operation as in previous releases. If the client provides a SOAP Action through the JAX-WS `javax.xml.ws.BindingProvider` properties, `SOAPACTION_USE_PROPERTY` and `SOAPACTION_URI_PROPERTY`, then that SOAP Action is used. Therefore, parsing of the outbound message does not occur regardless of the setting of the property. Setting a SOAP Action explicitly by the client is considered a best practice, particularly for performance on the service-provider. This practice prevents the inbound message from being parsed, to be routed to the correct endpoint operation.

Example

The following example illustrates the steps to create a Dispatch client and invoke a sample EchoService service endpoint.

```
String endpointUrl = ...;

QName serviceName = new QName("http://com/ibm/was/wssample/echo/",
 "EchoService");
QName portName = new QName("http://com/ibm/was/wssample/echo/",
 "EchoServicePort");

/** Create a service and add at least one port to it. **/
Service service = Service.create(serviceName);
service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING, endpointUrl);

/** Create a Dispatch instance from a service.**/
Dispatch<SOAPMessage> dispatch = service.createDispatch(portName,
SOAPMessage.class, Service.Mode.MESSAGE);

/** Create SOAPMessage request. **/
// compose a request message
MessageFactory mf = MessageFactory.newInstance(SOAPConstants.SOAP_1_1_PROTOCOL);

// Create a message.  This example works with the SOAPPART.
SOAPMessage request = mf.createMessage();
SOAPPart part = request.getSOAPPart();

// Obtain the SOAPEnvelope and header and body elements.
SOAPEnvelope env = part.getEnvelope();
SOAPHeader header = env.getHeader();
SOAPBody body = env.getBody();

// Construct the message payload.
SOAPElement operation = body.addChildElement("invoke", "ns1",
 "http://com/ibm/was/wssample/echo/");
SOAPElement value = operation.addChildElement("arg0");
value.addTextNode("ping");
request.saveChanges();

/** Invoke the service endpoint. **/
SOAPMessage response = dispatch.invoke(request);

/** Process the response. **/
```

Invoking JAX-WS web services asynchronously

Java API for XML-Based Web Services (JAX-WS) provides support for invoking web services using an asynchronous client invocation. JAX-WS provides support for both a callback and polling model when calling web services asynchronously. Both the callback model and the polling model are available on the Dispatch client and the Dynamic Proxy client.

Before you begin

Develop a JAX-WS Dynamic Proxy or Dispatch client. When developing Dynamic Proxy clients, after you generate the portable client artifacts from a Web Services Description Language (WSDL) file using the **wsimport** command, the generated service endpoint interface (SEI) does not have asynchronous methods included in the interface. Use JAX-WS bindings to add the asynchronous callback or polling methods on the interface for the Dynamic Proxy client. To enable asynchronous mappings, you can add the `jaxws:enableAsyncMapping` binding declaration to the WSDL file. For more information on adding binding customizations to generate an asynchronous interface, see chapter 8 of the JAX-WS specification.

Note: When you run the **wsimport** tool and enable asynchronous invocation through the use of the JAX-WS enableAsyncMapping binding declaration, ensure that the corresponding response message your WSDL file does not contain parts. When a response message does not contain parts, the request acts as a two-way request, but the actual response that is sent back is empty. The **wsimport** tool does not correctly handle a void response. To avoid this scenario, you can remove the output message from the operation which makes your operation a one-way operation or you can add a `<wsdl:part>` to your message. For more information on the usage, syntax and parameters for the **wsimport** tool, see the wsimport command for JAX-WS applications documentation.

About this task

An asynchronous invocation of a web service sends a request to the service endpoint and then immediately returns control to the client program without waiting for the response to return from the service. JAX-WS asynchronous web service clients consume web services using either the callback approach or the polling approach. Using a polling model, a client can issue a request and receive a response object that is polled to determine if the server has responded. When the server responds, the actual response is retrieved. Using the callback model, the client provides a callback handler to accept and process the inbound response object. The `handleResponse()` method of the handler is called when the result is available. Both the polling and callback models enable the client to focus on continuing to process work without waiting for a response to return, while providing for a more dynamic and efficient model to invoke web services. Polling invocations are valid from Enterprise JavaBeans (EJB) clients or Java Platform, Enterprise Edition (Java EE) application clients. Callback invocations are valid only from Java EE application clients.

Using the callback asynchronous invocation model

To implement an asynchronous invocation that uses the callback model, the client provides an `AsyncHandler` callback handler to accept and process the inbound response object. The client callback handler implements the `javax.xml.ws.AsyncHandler` interface, which contains the application code that is run when an asynchronous response is received from the server. The `javax.xml.ws.AsyncHandler` interface contains the `handleResponse(java.xml.ws.Response)` method that is called after the run time has received and processed the asynchronous response from the server. The response is delivered to the callback handler in the form of a `javax.xml.ws.Response` object. The response object returns the response content when the `get()` method is called. Additionally, if an error was received, then an exception is returned to the client during that call. The response method is then invoked according to the threading model used by the executor method, `java.util.concurrent.Executor` on the client's `java.xml.ws.Service` instance that was used to create the Dynamic Proxy or Dispatch client instance. The executor is used to invoke any asynchronous callbacks registered by the application. Use the `setExecutor` and `getExecutor` methods to modify and retrieve the executor configured for your service.

Using the polling asynchronous invocation model

Using the polling model, a client can issue a request and receive a response object that can subsequently be polled to determine if the server has responded. When the server responds, the actual response can then be retrieved. The response object returns the response content when the `get()` method is called. The client receives an object of type `javax.xml.ws.Response` from the `invokeAsync` method. That Response object is used to monitor the status of the request to the server, determine when the operation has completed, and to retrieve the response results.

Using an asynchronous message exchange

By default, asynchronous client invocations do not have asynchronous behavior of the message exchange pattern on the wire. The programming model is asynchronous; however, the exchange of request or response messages with the server is not asynchronous. To use an asynchronous message exchange, the `com.ibm.websphere.webservices.use.async.mep` property must be set on the client request context with a boolean value of `true`. When this property is enabled, the messages exchanged between the client and server are different from messages exchanged synchronously. With an asynchronous exchange, the request and response messages have WS-Addressing headers added that provide additional routing information for the messages. Another major difference between asynchronous and synchronous message exchange is that the response is delivered to an asynchronous listener that then delivers that response back to the client. For asynchronous exchanges, there is no timeout that is sent to notify the client to stop listening for a response. To force the client to stop waiting for a response, issue a `Response.cancel()` method on the object returned from a polling invocation or a `Future.cancel()` method on the object returned from a callback invocation. The cancel response does not affect the server when processing a request.

Procedure

1. Determine if you want to implement the callback method or the polling method for the client to asynchronously invoke the web service.
2. (Optional) Configure the client request context. Add the

```
com.ibm.websphere.webservices.use.async.mep
```

   property to the request context to enable asynchronous messaging for the web services client. Using this property requires that the service endpoint supports WS-Addressing which is supported by default for the application server. The following example demonstrates how to set this property:

```
Map<String, Object> rc = ((BindingProvider) port).getRequestContext();
 rc.put("com.ibm.websphere.webservices.use.async.mep", Boolean.TRUE);
```

3. To implement the asynchronous callback method, perform the following steps.
   a. Find the asynchronous callback method on the SEI or `javax.xml.ws.Dispatch` interface. For an SEI, the method name ends in *Async* and has one more parameter than the synchronous method of type `javax.xml.ws.AsyncHandler`. The `invokeAsync(Object, AsyncHandler)` method is the one that is used on the Dispatch interface.
   b. (Optional) Add the `service.setExecutor` methods to the client application. Adding the executor methods gives the client control of the scheduling methods for processing the response. You can also choose to use the `java.current.Executors` class factory to obtain packaged executors or implement your own executor class. See the JAX-WS specification for more information on using executor class methods with your client.
   c. Implement the `javax.xml.ws.AsyncHandler` interface. The `javax.xml.ws.AsyncHandler` interface only has the `handleResponse(javax.xml.ws.Response)` method. The method must contain the logic for processing the response or possibly an exception. The method is called after the client run time has received and processed the asynchronous response from the server.
   d. Invoke the asynchronous callback method with the parameter data and the callback handler.
   e. The `handleResponse(Response)` method is invoked on the callback object when the response is available. The `Response.get()` method is called within this method to deliver the response.
4. To implement the polling method,
   a. Find the asynchronous polling method on the SEI or `javax.xml.ws.Dispatch` interface. For an SEI, the method name ends in *Async* and has a return type of `javax.xml.ws.Response`. The `invokeAsync(Object)` method is used on the Dispatch interface.
   b. Invoke the asynchronous polling method with the parameter data.
   c. The client receives the object type, `javax.xml.ws.Response`, that is used to monitor the status of the request to the server. The `isDone()` method indicates whether the invocation has completed. When the `isDone()` method returns a value of `true`, call the `get()` method to retrieve the response object.
5. Use the `cancel()` method for the callback or polling method if the client needs to stop waiting for a response from the service. If the `cancel()` method is invoked by the client, the endpoint continues to process the request. However, the wait and response processing for the client is stopped.

Results

You have enabled your JAX-WS web service client to asynchronously invoke and consume web services. See the JAX-WS specification for additional information regarding the asynchronous client APIs.

Example

The following example illustrates a web service interface with methods for asynchronous requests from the client.

```
@WebService

public interface CreditRatingService {
       // Synchronous operation.
       Score getCreditScore(Customer     customer);
```

```
        // Asynchronous operation with polling.
        Response<Score> getCreditScoreAsync(Customer customer);
        // Asynchronous operation with callback.
        Future<?> getQuoteAsync(Customer customer,
               AsyncHandler<Score> handler);
}
```

Using the callback method

The callback method requires a callback handler that is shown in the following example. When using the callback procedure, after a request is made, the callback handler is responsible for handling the response. The response value is a response or possibly an exception. The `Future<?>` method represents the result of an asynchronous computation and is checked to see if the computation is complete. When you want the application to find out if the request is completed, invoke the `Future.isDone()` method. Note that the `Future.get()` method does not provide a meaningful response and is not similar to the `Response.get()` method.

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerTom,
        new AsyncHandler<Score>() {
               public void handleResponse (
                      Response<Score> response)
                   {
                      score = response.get();
                      // process the request...
                   }
    }
);
```

Using the polling method

The following example illustrates an asynchronous polling client:

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerTom);

while (!response.isDone()) {
        // Do something while we wait.
}

score = response.get();
```

Implementing extensions to JAX-WS web services clients

WebSphere Application Server provides extensions to web services clients using the Java API for XML-based Web Services (JAX-WS) programming model.

About this task

You can customize web services by using the following extensions to the JAX-WS client programming model.

Procedure

- Set the **JAXWS_OUTBOUND_SOAP_HEADERS** and **JAXWS_INBOUND_SOAP_HEADERS** properties on the request context of the Dispatch or Proxy object to enable a JAX-WS web services client to send or retrieve implicit SOAP headers.

  An implicit SOAP header is a SOAP header that is not explicitly defined in the WSDL file. An implicit SOAP header file fits one of the following descriptions:

  – A message part that is declared as a SOAP header in the binding in the WSDL file, but the message definition is not referenced by a portType within a WSDL file.

  – An element that is not contained in the WSDL file.

  Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

  To learn how to modify your client code to send or retrieve transport headers, see the information on sending implicit SOAP headers with JAX-WS or receiving implicit SOAP headers with JAX-WS.

- Set the **REQUEST_TRANSPORT_PROPERTIES** and **RESPONSE_TRANSPORT_PROPERTIES** properties to enable a web services client to send or retrieve transport headers.

Set the properties on the BindingProvider instance.

By modifying your client code to send or retrieve transport headers, you can send or receive specific information within the transport headers of outgoing requests or incoming responses from the server. For requests or responses that use the HTTP transport, the information is sent or retrieved in an HTTP header. Similarly, for a request or response that uses the Java Message Service (JMS) transport, the information is sent or retrieved in a JMS message property.

To learn how to modify your client code to send or retrieve transport headers, see the information on sending transport headers with JAX-WS or retrieving transport headers with JAX-WS.

To learn how to enable a web services client to send or retrieve transport headers, see the transport header properties best practices information.

Example: Using JAX-WS properties to manipulate SOAP headers in a JAX-WS handler

WebSphere Application Server provides extensions to the Java API for XML-Based Web Services (JAX-WS) and Web Services for Java Platform, Enterprise Edition (Java EE) client programming models, including the jaxws.binding.soap.headers.outbound and jaxws.binding.soap.headers.inbound properties. This example shows how to use these two properties to manipulate SOAP headers in a JAX-WS handler.

The following programming example illustrates how to set two request SOAP headers and retrieve one response SOAP header within a JAX-WS handler context:

```
1  //Create the hashmaps for the outbound soap headers
2  Map<QName, List<String>> outboundHeaders=messageContext.get("jaxws.binding.soap.headers.outbound");
3
4  //Add "AtmUuid1" and "AtmUuid2" to the outbound map
5  List<String> list1 = new ArrayList<String>();
6  list1.add("<AtmUuid1 xmlns=\"com.rotbank.security\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid1>");
7  List<String> list2 = new ArrayList<String>();
8  list2.add("<AtmUuid2 xmlns=\"com.rotbank.security\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid2>"
9  outboundHeaders.put(new QName("com.rotbank.security", "AtmUuid1"), list1);
10 outboundHeaders.put(new QName("com.rotbank.security", "AtmUuid2"), list2);
11 // Set the outbound map on the MessageContext object, which is passed into the JAX-WS handler method
12 messageContext.put("jaxws.binding.soap.headers.outbound", outboundHeaders);
```

On line 2, retrieve the outbound SOAP header map from the MessageContext parameter, which is passed into the JAX-WS handler method.

On lines 5-10, the AtmUuid1 and AtmUuid2 headers elements are added to the outbound map.

On line 12, the outbound map is set on the JAX-WS handler context, which causes the AtmUuid1 and AtmUuid2 headers to be added to the request message. This code is not necessary because the outboundHeaders map is a live map.

JAX-WS handler methods might also retrieve specific headers from the map and remove headers or entire lists of headers, if desired.

Sending implicit SOAP headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) web services client to send values in implicit SOAP headers. By modifying your client code to send implicit SOAP headers, you can send specific information within an outgoing web service request.

Before you begin

To complete this task, you need a web services client that you can enable to send implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

Using JAX-WS, there is no restriction on types of headers that you can manipulate.

About this task

The client application sets properties on the Dispatch or Proxy object to send and receive implicit SOAP headers.

Procedure

1. Create a `java.util.HashMap<QName, List<String>>` object.
2. Add an entry to the HashMap object for each implicit SOAP header that the client wants to send. The HashMap entry key is the QName of the SOAP header. The HashMap entry value is a `List<String>` object, and each String is the XML text of the entire SOAP header element. By using `List<String>` object, you can add multiple SOAP header elements that each have the same QName object.
3. Set the HashMap object as a property on the request context of the Dispatch or Proxy object. The property name is com.ibm.wsspi.websvcs.Constants.JAXWS_OUTBOUND_SOAP_HEADERS. The value of the property is the HashMap.
4. Issue the remote method calls using the Dispatch or Proxy object. The headers within the HashMap object are sent in the outgoing message.

   A WebServiceException error can occur if any of the following are true:

   - The HashMap object contains a key that is not a QName object or if the HashMap object contains a value that is not a `List<String>` object.
   - The String representing an SOAP header is not a compliant XML message.
   - The HashMap contains a key that represents a SOAP header that is declared protected by the owning component.

Results

You have a JAX-WS web services client that is configured to send implicit SOAP headers.

Example

The following programming example illustrates how to set two request SOAP headers and retrieve one response SOAP header within a JAX-WS web services request and response context:

```
1  //Create the hashmaps for the outbound soap headers
2  Map<QName, List<String>> outboundHeaders=new HashMap<QName, List<String>>();
3
4  //Add "AtmUuid1" and "AtmUuid2" to the outbound map
5  List<String> list1 = new ArrayList<String>();
6  list1.add("<AtmUuid1 xmlns=\"com.rotbank.security\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid1>");
7  List<String> list2 = new ArrayList<String>();
8  list2.add("<AtmUuid2 xmlns=\"com.rotbank.security\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid2>"
9  outboundHeaders.put(new QName("com.rotbank.security", "AtmUuid1"), list1);
10 outboundHeaders.put(new QName("com.rotbank.security", "AtmUuid2"), list2);
11 // Set the outbound map on the request context
12 dispatch.getRequestContext().put("jaxws.binding.soap.headers.outbound");
13 // Invoke the remote operation
14 dispatch.invoke(parm1);
15 // Get the inbound header map from the response context
16 Map<QName,List<String>> inboundMap = dispatch.getResponseContext().get("jaxws.binding.soap.headers.outbound");
17 List<String> serverUuidList = inboundMap.get(new QName("com.rotbank.security","ServerUuid"));
18 String text = serverUiidList.get(0);
19 //Note: text now equals a XML object that contains a SOAP header:
21//"<y:ServerUuid xmlns:y=\"com.rotbank.security\"><:uuid>ROTB-0A03519322FSA01
22  </y:uuid></y:ServerUuid.");
```

On line 2, create the outbound SOAP header map.

On lines 5-10, the AtmUuid1 and AtmUuid2 headers elements are added to the outbound map.

On line 12, the outbound map is set on the request context, which causes the AtmUuid1 and AtmUuid2 headers to be added to the request message when the operation is invoked.

On line 14, invoke the remote operation.

On line 15, obtain the outbound header map.

On lines 17-18, the ServerUuid header is retrieved from the response Map. The Map accesses the SOAP header from the response message.

Receiving implicit SOAP headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) web services client to receive values from implicit SOAP headers. By modifying your client code to receive implicit SOAP headers, you can receive specific information within an incoming web service response.

Before you begin

To complete this task, you need a web services client that you can enable to receive implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

Using JAX-WS, there is no restriction on types of headers that you can manipulate.

About this task

The client application sets properties on the Dispatch or Proxy object to send and receive implicit SOAP headers.

Procedure

1. Issue a remote method call with the Dispatch or Proxy object.
2. Using the property name, `com.ibm.wsspi.websvcs.Constants.JAXWS_INBOUND_SOAP_HEADERS`, retrieve the `Map<QName, List<String>>` from the ResponseContext of the Dispatch or Proxy object.
3. From the `Map<QName, List<String>>` value, retrieve a `List<String>` using the QName of the SOAP header. If the `List<String>` value is present, that value contains zero or more String objects that contain the XML text of the SOAP headers for the corresponding QName.

Results

You have a JAX-WS web services client that can receive values from implicit SOAP headers.

Transport header properties best practices

You can set the REQUEST_TRANSPORT_PROPERTIES property and RESPONSE_TRANSPORT_PROPERTIES property on a Java API for XML-based RPC (JAX-RPC) client Stub, a Call instance, or a Java API for XML-Based Web services (JAX-WS) BindingProvider's RequestContext instance to enable a web services client to send or retrieve transport headers.

Note: Use these best practices to enable a Web services client to send or retrieve transport headers.

REQUEST_TRANSPORT_PROPERTIES best practices

Some transport headers such as the HTTP Cookie header and the Cookie2 header contain multiple embedded values. For headers that contain multiple values, the header value must be written in the following way:

- Each name=value pair embedded within the header value must be separated by a semi-colon (;).
- Each *name* and its value must be separated by an equal (=) sign.

The following is an example of how the header value must be written:

```
name1=value1;name2=value2;name3=value3
```

The values contained in the user's Map might be parsed before being added to the outgoing request if the outgoing request already contains a header identifier that matches one in the Map. For certain transport headers that contain multiple embedded values, the header values in the Map are parsed into individual name=value components. A semi-colon (;) separates the components, for example, name1=value1;name2=value2. Each name=value is appended to the outgoing header unless:

- The outgoing request header contains a *name* value.

  In this case, the name=value from the Map is silently ignored, preventing a client from overwriting or modifying values for the *name* value that are already set in the outgoing request header by either the server or the web services engine.
- The user's header value contains multiple *name* values.

  When the user's header value contains multiple *name* values, the first occurrence of the *name* value is used and the others are silently ignored. For example, if the user's header value contains name1=value1;name2=value2;name1=value3, where there are two occurrences of name1, the first value, name1=value1, is used. The other value, name1=value3, is silently ignored.

RESPONSE_TRANSPORT_PROPERTIES best practices

Only the Map keys are used; the Map values are ignored. The values are filled in this Map by retrieving the transport headers, which correspond to the Map keys from the incoming response message. An empty Map causes all of the transport headers and the associated values to be retrieved from the incoming response message.

HTTP headers that are processed under special consideration

The following are HTTP headers that are given special consideration when sending and retrieving HTTP responses and requests.

The values in these headers can be set in a variety of ways. For example, some header values are sent based on settings in a deployment descriptor or binding file. In these cases, the value set through REQUEST_TRANSPORT_PROPERTIES overrides the values set any other way.

Table 131. HTTP request and response header values. Values to specify for HTTP headers when sending and retrieving HTTP responses and requests.

Header	Send request	Retrieve response
Transfer-encoding	• The transfer-encoding header is ignored for HTTP 1.0. • When using HTTP 1.1, the transfer-encoding header is set to `chunked` if the value is `chunked`.	There is no special processing.

Table 131. HTTP request and response header values (continued). Values to specify for HTTP headers when sending and retrieving HTTP responses and requests.

Header	Send request	Retrieve response
Connection	• The connection header is ignored for HTTP 1.0. • When using HTTP 1.1, the following values are set: – The connection header is set to "close" if the value is set to "close". – The connection header is set to "keep-alive" if the value is set to "keep-alive". – All other value settings are ignored.	There is no special processing.
Expect	• The expect header is ignored for HTTP 1.0. • When using HTTP 1.1, the following values are set: – The connection header is set to "100-continue" if the value is set to "100-continue". – All other value settings are ignored.	There is no special processing.
Host	Ignored	There is no special processing.
Content-type	Ignored	There is no special processing.
SOAPAction	Ignored	There is no special processing.
Content-length	Ignored	There is no special processing.
Cookie The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_COOKIE	The value is sent on the header if it is structured correctly. See the information in this article for Header value format and Map values.	There is no special processing.
Cookie2 The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_COOKIE2	See the information in the "Cookie" entry.	There is no special processing.
Set-cookie The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_SET_COOKIE	There is no special processing.	If the property MAINTAIN_SESSION is set to `true`, the entire value is saved into SessionContext.CONTEXT_PROPERTY and is sent on subsequent requests in the Cookie header. See the Cookie entry in this table for more information.
Set-cookie2 The following is a String constant: com.ibm.websphere.webservices.Constants .HTTP_HEADER_SET_COOKIE2	There is no special processing.	If the property MAINTAIN_SESSION is set to `true`, the entire value is saved into SessionContext.CONTEXT_PROPERTY and is sent on subsequent requests in the Cookie header. See the Cookie entry in this table for more information.

Example client code

The following is an example of how you can code a web services client to send and retrieve transport header values:

```
public class MyApplicationClass {
    // Inject an instance of the service's port-type.
    @WebServiceRef(EchoService.class)
    private EchoPortType port;

    // This method will invoke the web service operation and send and receive transport headers.
    public void invokeService() {

        // Set up the Map that will contain the request headers.
        Map<String, Object>requestHeaders = new HashMap<String, Object>();
        requestHeaders.put("Cookie", "ClientAuthenticationToken=FFEEBBCC");
        requestHeaders.put("MyHeaderFlag", new Boolean(true));
```

```
        // Set the Map as a property on the RequestContext.
        BindingProvider bp = (BindingProvider) port;
        bp.getRequestContext().put(com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES, requestHeaders);

        // Set up the Map to retrieve transport headers from the response message.
        Map<String, Object>responseHeaders = new HashMap<String, Object>();
        responseHeaders.put("Set-Cookie", null);
        responseHeaders.put("MyHeaderFlag, null);

        // Invoke the web services operation.
        String result = port.echoString("Hello, world!");

        // Retrieve the headers from the response.
        String cookieValue = responseHeaders.get("Set-Cookie");
        String headerFlag = responseHeaders.get("MyHeaderFlag");
    }
}
```

Sending transport headers with JAX-WS

You can enable an existing Java API for XML-Based Web Services (JAX-WS) web services client to send application-defined information along with your web services requests by using transport headers. In addition, you can enable a JAX-WS Web services endpoint to send application-defined information along with the Web services response message by using transport headers.

Before you begin

You need a JAX-WS web services client that you can enable to send transport headers.

Sending transport headers is supported in JAX-WS Web services clients, and is supported for the HTTP and JMS transports. The web services client must call the JAX-WS APIs directly and not through any intermediary layers, such as a gateway function.

About this task

When using the JAX-WS programming model, the client must set a property on the BindingProvider's RequestContext object to send values in transport headers with the Web services request message. After you set the property, the values are set in all the requests for subsequent remote method invocations against the BindingProvider object until the associated property is set to `null` or the BindingProvider object is discarded.

To send values in the transport headers on outbound requests from a JAX-WS Web services client application, modify the client code as follows:

Procedure

1. Create a java.util.Map object that contains the transport headers.
2. Add an entry to the Map object for each transport header that you want the client to send.
   a. Set the Map entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user-defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made as to the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class.
   b. Set the Map entry value to the value of the transport header. The type of this value can be one of the following data types:
      - java.lang.String
      - java.lang.Integer
      - java.lang.Short
      - java.lang.Long
      - java.lang.Float

- java.lang.Double
- java.lang.Byte
- java.lang.Boolean

3. Set the Map object on the BindingProvider's RequestContext using the com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES property. When the REQUEST_TRANSPORT_PROPERTIES property value is set, that Map is used on subsequent invocations to set the header values in the outgoing requests. If the REQUEST_TRANSPORT_PROPERTIES property value is set to `null`, no transport properties are sent in outgoing requests. To learn more about these properties, see the transport header properties documentation.
4. Issue remote method calls against the BindingProvider instance. The headers and the associated values from the Map are added to the outgoing request for each method invocation. If the invocation uses HTTP, then the transport headers are sent as HTTP headers within the HTTP request. If the invocation uses JMS, then the transport headers are sent as JMS message properties.

   If the Constants.REQUEST_TRANSPORT_PROPERTIES property is not set correctly, you might experience API usage errors that result in a WebServiceException error. The following requirements must be met, or the process fails:

   a. The Constants.REQUEST_TRANSPORT_PROPERTIES property value that is set on the BindingProvider's RequestContext must be a java.util.Map object or `null`.
   b. Each key in the Map must be a java.util.String data type.
   c. Each value in the Map must be one of the following data types:
      - java.lang.String
      - java.lang.Integer
      - java.lang.Short
      - java.lang.Long
      - java.lang.Float
      - java.lang.Double
      - java.lang.Byte
      - java.lang.Boolean

Results

You have a JAX-WS web services client that is configured to send transport headers in Web services request messages.

Example

Here is a short programming example that illustrates how request transport headers are sent by a JAX-WS Web services client application:

```
public class MyApplicationClass {
    // Inject an instance of the service's port-type.
    @WebServiceRef(EchoService.class)
    private EchoPortType port;

    // This method will invoke  the web service operation and send transport headers on the request.
    public void invokeService() {

        // Set up the Map that will contain the request headers.
        Map<String, Object> requestHeaders = new HashMap<String, Object>();
        requestHeaders.put("MyHeader1", "This is a string value");
        requestHeaders.put("MyHeader2", new Integer(33));
        requestHeaders.put("MyHeader3", new Boolean(true));

        // Set the Map as a property on the RequestContext.
        BindingProvider bp = (BindingProvider) port;
        bp.getRequestContext().put(com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES, requestHeaders);
```

```
        // Invoke the web services operation.
        String result = port.echoString("Hello, world!");
    }
}
```

Sending response transport headers from a JAX-WS endpoint:
Before you begin

You need a JAX-WS Web services endpoint implementation class that you can enable to send transport headers.

Sending response transport headers from a JAX-WS endpoint is similar to sending request transport headers from a JAX-WS client. It is supported for the HTTP and JMS transports.

About this task

When using the JAX-WS programming model, the endpoint implement class must set a property on the MessageContext to send values in transport headers with the Web services response message.

Procedure

1. Create a java.util.Map object that contains the transport headers.
2. Add an entry (key and value) to the Map object for each transport header that you want to send in the response message. This is similar to the procedure for the client above.
3. Retrieve the MessageContext (instance of `javax.xml.ws.handler.MessageContext`) associated with the invocation of the endpoint.
4. Set the Map object on the MessageContext using the com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES property.

Results

You have a JAX-WS Web services endpoint implementation class that is configured to send transport headers in Web services response messages.

Example

Here is a short programming example that illustrates how response transport headers are sent by a JAX-WS Web services endpoint implementation class:

```
@WebService
public class EchoServiceImpl implements EchoServicePortType {

    // Inject an instance of WebServiceContext so we can retrieve
    // the MessageContext for each invocation of this endpoint.
    @Resource
    WebServiceContext ctxt;

    /**
     * Default constructor.
     */
    public EchoServiceImpl() {
        ....
    }

    public String echoString(String input) {
        String result = "Echo result: " + input;

        // Retrieve the MessageContext from the injected WebServiceContext.
        MessageContext mc = ctxt.getMessageContext();

        // Send some headers back in the response message.
        Map<String, Object> responseHeaders = new HashMap<String, Object>();
        responseHeaders.put("MyHeader1", "This is a string response value");
        responseHeaders.put("MyHeader2", new Integer(33));
        responseHeaders.put("MyHeader3", new Boolean(false));
```

```
        // Set the response header Map on the MessageContext.
        mc.put(com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES, responseHeaders);

        return result;
    }
}
```

Retrieving transport headers with JAX-WS

You can enable a Java API for XML-Based Web Services (JAX-WS) web services client to retrieve values from transport headers. For a request that uses HTTP, the transport headers are retrieved from HTTP headers found in the HTTP response message. For a request that uses Java Message Service (JMS), the transport headers are retrieved from the JMS message properties found on the JMS response message.

Before you begin

You need a JAX-WS web services client that you can enable to retrieve transport headers.

Retrieving transport headers is supported only by Web services clients, and is supported for the HTTP and JMS transports. The web services client must call the JAX-WS APIs directly and not through any intermediary layers, such as a gateway function.

About this task

When using the JAX-WS programming model, the client must set a property on the BindingProvider's RequestContext object to retrieve values from the transport headers. After you set the property, values are read from responses for the subsequent method invocations against that BindingProvider object until the associated property is set to `null` or the BindingProvider object is discarded.

To retrieve values from the transport headers on inbound responses, modify the client code as follows:

Procedure

1. Create a java.util.Map object that will hold the transport headers retrieved from the response message. To retrieve all the transport headers from a response message, leave this Map empty.
2. (Optional) Add an entry to the Map for each header that you want to retrieve from the incoming response message.
   a. Set the Map entry key to a string that exactly matches the transport header identifier. You can specify the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user-defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made to confirm the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class. The Map entry value is ignored and does not need to be set. An empty Map, for example, one that is non-null, but does not contain any keys, causes all the transport headers in the response to be retrieved.
3. Set the Map object on the BindingProvider's RequestContext using the com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES property. When the Map is set, the RESPONSE_TRANSPORT_PROPERTIES property is used in subsequent invocations to retrieve the headers from the responses. If you set the property to `null`, no headers are retrieved from the response. To learn more about these properties, see the transport header properties documentation.
4. Invoke remote method calls against the BindingProvider instance. The values from the specified transport headers are retrieved from the response message and placed in the Map.

   If the property is not set correctly, you might experience API usage errors that result in a WebServiceException error. The following requirements must be met, or the process fails:

- The Constants.RESPONSE_TRANSPORT_PROPERTIES property value that is set on the BindingProvider's RequestContext instance must be either `null` or an instance of java.util.Map.
- All the Map keys must be of the java.lang.String data type, and the keys must not be null.
- The Map may be empty, which means that it contains no entries at all. In this case, all the transport headers will be retrieved from the response message.

Results

You have a JAX-WS web service that can receive transport headers from incoming response messages.

Example

Here is a short programming example that illustrates how response transport headers are retrieved by a JAX-WS Web services client application:

```
public class MyApplicationClass {
    // Inject an instance of the service's port-type.
    @WebServiceRef(EchoService.class)
    private EchoPortType port;

    // This method will invoke  the web service operation and retrieve transport headers on the request.
    public void invokeService() {

        // Set up the Map to retrieive our response headers.
        Map<String, Object> responseHeaders = new HashMap<String, Object>;
        responseHeaders.put("MyHeader1", null);
        responseHeaders.put("MyHeader2", null);
        responseHeaders.put("MyHeader3", null);

        // Set the Map as a property on the RequestContext.
        BindingProvider bp = (BindingProvider) port;
        bp.getRequestContext().put(com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES, responseHeaders);

        // Invoke the web services operation.
        String result = port.echoString("Hello, world!");

        // Now retrieve our response headers.
        Object header1 = responseHeaders.get("MyHeader1");
        Object header2 = responseHeaders.get("MyHeader2");
        Object header3 = responseHeaders.get("MyHeader3");

    }
}
```

Developing JAX-RPC web services

Setting up a development environment for web services

The application server provides command-line tools to develop web services clients and implementations that are based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. You must set up your development environment before you start developing web services.

Before you begin

Before you can set up a web services development environment within WebSphere Application Server, you must install WebSphere Application Server. For detailed information on installing the application server, read about installing your application server environment.

About this task

Set up a web services development environment by completing the following actions.

Procedure

1. Set up the environment.
   Run the **setupCmdLine** script from the */profile_root*/<*application_server*>/bin directory.
2. Configure the path. You can add the WebSphere and Java bin directories to your path by typing:

```
set PATH=%WAS_PATH%;%PATH%
```

Results

You have set up an environment so that you can develop Web services.

What to do next

Implement web services applications. See the task overview for implementing web services applications information to learn about how to develop and implement a Java EE web service.

Developing a service endpoint interface from JavaBeans for JAX-RPC applications

You must develop a service endpoint interface if you are developing a JAX-RPC web service from a JavaBeans implementation.

Before you begin

You need to set up a development environment for web services and access an existing Java bean web application archive (WAR) file. See the setting up a development environment for web services information.

About this task

This task is a required step in developing a JAX-RPC Web service from a Java bean.

The service endpoint interface defines the methods for particular Java API for XML-based RPC (JAX-RPC) web services. The JavaBeans implementation must implement methods with the same signature as the methods on the service endpoint interface. A number of restrictions apply on which types to use as parameters and results of service endpoint interface methods. These restrictions are documented in the JAX-RPC specification.

You can also create a service endpoint interface by using assembly tools.

Develop a service endpoint interface for a JavaBeans implementation by following the actions listed:

Procedure

1. Create a Java interface that contains the methods to include in the service endpoint interface. If you start with an existing Java interface, remove any methods that do not conform to the JAX-RPC specification.
2. Compile the interface.
   Use the name of the service endpoint interface class in the **javac** command for the class to compile.

Results

You have developed a service endpoint interface that you can use to develop web services.

Example

The following example depicts the AddressBook interface:

```
package addr;
public interface AddressBook {
    /**
     * Retrieve an entry from the AddressBook.
     *
     *@param name the name of the entry to look up.
     *@return the AddressBook entry matching name or null if none.
     *@throws java.rmi.RemoteException if communications failure.
     */
    public addr.Address getAddressFromName(java.lang.String name);
}
```

Use the `AddressBook` interface to create the service endpoint interface:

1. Make a copy of the `AddressBook.java` interface and name it `AddressBook_SEI.java`. Use this copy as a template for the service endpoint interface.
2. Compile the interface.

What to do next

Continue to gather the artifacts that are required to develop a web service, including the Web Services Description Language (WSDL) file. You need to develop a WSDL file because it is the engine of a web service. Without a WSDL file, you do not have a web service. See the developing a WSDL file for JAX-RPC applications information.

Developing a service endpoint interface from enterprise beans for JAX-RPC applications

You can develop a service endpoint interface from an Enterprise JavaBeans (EJB) for JAX-RPC web services.

Before you begin

Set up a development environment for web services. To learn more, see the setting up a development environment for web services information.

Set up a development environment for web services.

This task is a required step in developing a Java API for XML-based RPC (JAX-RPC) web service from an enterprise bean.

The service endpoint interface defines the web services methods. The enterprise beans that implements the web service must implement methods having the same signature as the methods of the service endpoint interface. A number of restrictions exist on which types to use as parameters and results of service endpoint interface methods. These restrictions are documented in the Java API for XML-based remote procedure call (JAX-RPC) specification. See the web services specifications and API documentation to review the JAX-RPC specification along with a complete list of the supported standards and specifications.

The easiest method for creating the service endpoint interface for an EJB web service implementation is from the EJB remote interface.

You can also create a service endpoint interface by using assembly tools..

About this task

Develop a service endpoint interface by following the steps provided in this task section.

Procedure

1. Create a Java interface that contains the methods that you want to include in the service endpoint interface. If you start with an existing Java interface, remove any methods that do not conform to the JAX-RPC specification.
2. Compile the interface.

   Use the name of the service endpoint interface class in the **javac** command for the class to compile.

Results

You have a service endpoint interface that you can use to develop a web service.

Example

This example uses the EJB remote interface, `AddressBook_RI`, to create a service endpoint interface for an EJB implementation that is used as a web service. The following code example illustrates the AddressBook_RI remote interface.

```
package addr;
public interface AddressBook_RI extends javax.ejb.EJBObject {
    /**
     * Retrieve an entry from the AddressBook.
     *
     *@param name the name of the entry to look up.
     *@return the AddressBook entry matching name or null if none.
     *@throws java.rmi.RemoteException if communications failure.
     */
    public addr.Address getAddressFromName(java.lang.String name)
     throws java.rmi.RemoteException;
}
```

Use the following steps to create the service endpoint interface with the `AddressBook_RI` remote interface:

1. Locate a remote interface that has already been created, like the `AddressBook_RI.java` remote interface.
2. Make a copy of the `AddressBook.java` remote interface and use it as a template for the service endpoint interface.
3. Compile the `AddressBook.java` service endpoint interface.

What to do next

Continue gathering the artifacts that are required to develop a web service, including the Web Services Description Language (WSDL) file. You need to develop a WSDL file because it is the engine of a web service; without a WSDL file, you have no web service.

Developing a WSDL file for JAX-RPC applications

You can develop a Web Services Description Language (WSDL) file to describe the characteristics of your Java API for XML-based RPC (JAX-RPC) web services application including where the service resides and how to invoke the service using an XML format.

Before you begin

Depending on your development path, develop a service endpoint interface from a JavaBeans implementation or develop a service endpoint interface from an enterprise bean implementation.

About this task

You need a WSDL file to use web services. You can develop your own WSDL file or get one from a web services provider through email, downloading, or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own.

Develop a WSDL file by following the actions listed:

Procedure

1. Configure the service endpoint interface class and referenced classes into your CLASSPATH variable.
   - On Windows systems, set `CLASSPATH="%CLASSPATH%;<list your application Java archive (JAR) files and classes>"`.
   - On UNIX and Linux systems, export `CLASSPATH="$CLASSPATH:<list your application JAR files and classes>"`.
2. Run the **Java2WSDL** *seiInterface* command. A WSDL file named *seiInterface*`.wsdl` is created.

   Note: The **Java2WSDL** command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the **Java2WSDL** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.
   - Move the WSDL file to the `META-INF/wsdl` subdirectory if you are using Enterprise JavaBeans (EJB).
   - Move the WSDL file to the `WEB-INF/wsdl` subdirectory if you are using JavaBeans.
3. Edit the generated WSDL file and inspect the part names. The WSDL parts have names like `arg_0_0`. Modify the WSDL file to use the actual names of the Java parameters.
4. (Optional) Use the **Java2WSDL** command tool to generate the correct part names of WSDL file. You can automatically generate and set the correct part names by using the **Java2WSDL** command tool. Generating and setting the part names is done by providing additional information to the **Java2WSDL** command tool in the form of a Java implementation class that implements the same methods as the service endpoint interface and is compiled with debug information turned on. Parameter names are stored in the `.class` file with the debug information. If your implementation class is compiled with debug on, you can use the `Java2WSDL -implClass` *seiImpl seiInterface* command to generate a WSDL file with the proper part names.

Results

A WSDL file that defines the web services described by the service endpoint interface.

Example

This example uses the JAR file name `AddressBook.jar` that contains a class named `AddressBook.class` class file.

You must add the `AddressBook.jar` file to your CLASSPATH to create the WSDL file. The JAR file contains an EJB implementation class that is compiled with debugging information turned on. Run the `Java2WSDL -implClass addr.AddressBookBean addr.AddressBook` command to create the file, `AddressBook.wsdl`.

What to do next

Depending on your development path, develop web services deployment descriptor templates for JavaBeans or develop web services deployment descriptor templates for an enterprise beans implementation.

Java2WSDL command for JAX-RPC applications

The Java2WSDL command-line tool maps Java classes to a WSDL file for Java API for XML-based RPC (JAX-RPC) applications.

The Java2WSDL command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product.

The **Java2WSDL** command maps a Java class to a Web Services Description Language (WSDL) file by following the Java API for XML-based Remote Procedure Call (JAX-RPC) 1.1 specification.

The **Java2WSDL** command accepts a Java class as input and produces a WSDL file that represents the input class. If a file exists at the output location, it is overwritten. The WSDL file that is generated by the **Java2WSDL** command contains WSDL and XML schema constructs that are automatically derived from the input class. You can override these default values with command-line arguments.

The **Java2WSDL** command is protocol independent; when you run the **Java2WSDL** command, you can specify command-line options that generate both SOAP and non-SOAP protocol bindings in the WSDL file. For each binding that can be generated, the **Java2WSDL** command has a binding generator to generate the WSDL for that binding.

Command line syntax and arguments

The command line syntax is:

```
Java2WSDL [argument...] class
```

The following command-line arguments are supported:

Required arguments

- class

  Represents the fully qualified name of one of the following Java classes:
  - Stateless session Enterprise JavaBeans (EJB) remote interface that extends the javax.ejb.EJBObject class
  - Service endpoint interface that extends the java.rmi.Remote class
  - Java beans

  The **Java2WSDL** command locates the class in the CLASSPATH variable.

Important arguments

- -location *location*

  Provides the published location or the Uniform Resource Locator (URL) of the service. If this information is not provided, a warning is issued that indicates that the final published location is not determined yet. The service location is typically overridden when the web service is deployed.

  The name after the last backslash is the name of the service port, unless the name is overridden by the -servicePortName argument. The service port address location attribute is assigned the specified value. Multiple endpoint addresses can be specified. Using the -location option is recommended only if a single binding type is required. If multiple binding types are requested, protocol binding-specific location properties are passed over the command line using the -x flag.

  The following example illustrates how to produce both SOAP over HTTP, and SOAP over Java Message Service (JMS) bindings :

```
java2wsdl -bindingTypes http,jms \
  -x http.location=http://your.server.name:9080/StockQuoteService/services/StockQuote \
  -x jms.location= \
 jms:/queue?destination=jms/MyQueue&connectionFactory=jms/MyCF&targetService=StockQuote
```

  Use the -location option to determine to which port the -location option value applies by requiring the endpoint URLs to be specified through the binding-specific property values.
- -output *wsdl-uri*

  Indicates the path and file name of the output WSDL file. If not specified, the default `class.wsdl` file is written into the current directory.
- -input *wsdl-uri*

  Specifies the input WSDL file that is used to build an output WSDL file. Information from an existing WSDL file, is specified in this option and is used with the input Java class to generate the output.
- -bindingTypes

Specifies the list of binding types write to the output WSDL file. Each binding generator in the **Java2WSDL** command supports specific binding types. The valid binding type values are `http` (SOAP over HTTP), `jms` (SOAP over JMS) and `ejb` (local or remote EJB invocation). For example, the following command can be used to generate SOAP over HTTP, EJB bindings for the my.pkg.MySEI Service Endpoint Interface and the my.pkg.MyEJBClass implementation class :

```
java2wsdl -bindingTypes http,ejb -implClass my.pkg.MyEJBClass my.pkg.MySEI
```

The following command is an example of using the -bindingTypes option to generate SOAP over HTTP and SOAP over JMS bindings:

```
java2wsdl -bindingTypes http,jms -implClass my.pkg.MyEJBClass my.pkg.MySEI
```

- -style RPC | DOCUMENT

  Specifies the WSDL style to use in the generated WSDL file. To learn more, see the Mapping between Java language, WSDL and XML for JAX-RPC applications information. This argument is used with the -use argument.

  If RPC is specified with -use ENCODED, a `style=rpc/use=encoded` WSDL file is generated. If RPC is specified with the -use LITERAL option, a `style=rpc/use=literal` WSDL file is generated. If DOCUMENT is specified with the -use LITERAL option, a `style=document/use=literal` WSDL file is generated.
- -use LITERAL | ENCODED

  Specifies which style and use combinations are generated into the WSDL file when used with the -style argument. The combinations are rpc and encoded, rpc and literal, or doc and literal. This setting applies to all SOAP bindings. To learn more, see the Mapping between Java language, WSDL and XML for JAX-RPC applications information.
- -transport http | jms

  Generates SOAP bindings for either HTTP (default) or JMS. If JMS is specified, the characters `jms` are appended to the WSDL file name to prevent overwriting an existing WSDL file for another transport. The transport option can be specified only once.

  This option is deprecated. The -bindingTypes option replaces the -transport option, so that you can generate bindings that are non-SOAP specific.
- -portTypeName *name*

  Specifies the name to use for the portType element. If not specified, the binding name is the port type name.
- -bindingName *name*

  Specifies the name to use for the binding element. If not specified, the binding name is the port type name.
- -serviceElementName *name*

  Specifies the name of the service element.
- -servicePortName *name*

  Specifies the name of the service. If not specified, the service name is derived from the -location argument.
- -namespace *targetNamespace*

  Indicates the target namespace for the WSDL file being generated. To learn about the algorithm that is used to obtain the default namespace, see the Mapping between Java language, WSDL and XML for JAX-RPC applications information.
- -PkgtoNS *package namespace*

  Specifies the mapping of a Java package to a namespace. If a package does not have a namespace, the **Java2WSDL** command generates a namespace name. You can repeat the -PkgtoNS argument to specify mappings for multiple packages.
- -extraClasses *classes*

  Specifies other classes that are represented in the WSDL file.
- -implClass *impl-class*

  The **Java2WSDL** command uses method parameter names to construct the WSDL file message part names. The command automatically obtains the message names from the debug information in the

class. If the class is compiled without debug information, or if the class is an interface, the method parameter names are not available. In this case, you can use the -implClass argument to provide an alternative class from which to obtain method parameter names. The impl-class does not need to implement the class if the class is an interface, but it must implement the same methods as the class.

- -verbose

  Displays verbose messages.
- -help

  Displays the help message.
- -helpX

  Displays the help message for extended options and for various options that are supported by binding generators.

Other arguments

- -wrapped *boolean*

  Specifies whether to generate the WSDL file according to wrapped rules. This option is valid if `use` is literal only. The option defaults to `true`.
- -stopClasses *parent* [, *parent*]

  The **Java2WSDL** command searches inherited classes and interfaces to construct the list of methods for WSDL file operations if the -all argument is specified.

  The **Java2WSDL** command searches inherited classes and interfaces when generating extended complexTypes. The search stops when a class or an interface is found within a package that begins with `java` or `javax`. You can use the -stopClasses argument to define additional classes that cause the search to stop.
- -methods *argument*

  Specifies a list of method names from the Service Endpoint Interface that must be exposed in the output WSDL file. The list is separated by spaces or commas.
- -soapAction

  Valid arguments are:
  - DEFAULT

    Sets the soapAction field according to the deployment information.
  - NONE

    Sets the soapAction field to double quotes ("").
  - OPERATION

    Sets the soapAction field to the operation name.
- -outputImpl *impl-wsdl*

  Specifies if you want an interface and implementation WSDL file emitted.
- -locationImport *location-uri*

  Specifies the location of the interface WSDL file if you use the -outputImpl argument.
- -namespaceImpl *namespace*

  Specifies the target namespace for the implementation WSDL file, if you use the -outputImpl argument.
- -MIMEStyle <*style*>

  Specifies the Multipurpose Internet Mail Extensions (MIME)- type used to map to Web Services-Interoperability (WS-I) SOAP with attachments reference (wsi:swaRef) for the binding element.

  <*style*> can be one of the following:
  - WSDL11 (default): Exclusively map MIME types using WSDL 1.1 standards. If the MIME type cannot map to WSDL 1.1 standards, the command fails.
  - AXIS: Map MIME types using AXIS standards, for example image becomes axis:image.
  - swaRef: Map MIME types using WSDL 1.1 standards with two caveats:
    - DataHandler maps to the wsi:swaRef element instead of an application and octet-stream

- If mapping is illegal through WSDL 1.1, map to the wsi:swaRef element

- -propertiesFile *argument*

  Sets existing options, such as -extraClasses, with a properties file instead of with a command line. The following example illustrates the use of this argument:

```
extraClasses=com.ibm.Class1, com.sun.Class2,org.apache.Class3
```

- -voidReturn

  Valid arguments are:
  – ONEWAY

    Methods with void returns are one-way. This argument is the default for a JMS transport.
  – TWOWAY

    Methods with void returns are two-way. This argument is the default for an HTTP transport.

- -debug

  Displays debug messages.

- -property or -x

  You can use the -x option to pass command-line options to various binding generators. Use the -x option multiple times on the command line to specify a set of property values to pass to each binding generator method called by the **Java2WSDL** command. You can also use a single -x option to specify multiple properties by separating them with a comma, for example:

```
java2wsdl -x prop1=value1 -x prop2=value2
```

  is equivalent to:

```
java2wsdl -x prop1=value1,prop2=value2
```

  The -x option provides flexibility to specify each command-line option for each binding generator individually, if required. The value specified in the -x option overrides the value that is specified in the equivalent command-line option if both are specified.

Mapping between Java language, WSDL and XML for JAX-RPC applications

Data for Java API for XML-based Remote Procedure Call (JAX-RPC) applications flows as extensible Markup Language (XML). JAX-RPC applications use mappings to describe the data conversion between the Java language and extensible Markup Language (XML) technologies, including XML Schema, Web Services Description Language (WSDL) and SOAP that are supported by the application server.

For JAX-RPC applications, most of the mappings between the Java language and XML are specified by the JAX-RPC specification. Some mappings that are optional or unspecified in JAX-RPC are also supported. Review the JAX-RPC specification for a complete list of APIs. For a complete list of the supported standards and specifications, see the web services specifications and API documentation.

Notational conventions

Table 132. Namespace conventions. Describes the namespace prefixes and the corresponding namespace used in namespace conventions.

Namespace prefix	Namespace
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
soapenc	http://schemas.xmlsoap.org/soap/encoding/
wsdl	http://schemas.xmlsoap.org/wsdl/
wsdlsoap	http://schemas.xmlsoap.org/wsdl/soap/
ns	user-defined namespace
apache	http://xml.apache.org/xml-soap
wasws	http://websphere.ibm.com/webservices/

Detailed mapping information

The following sections identify the supported mappings, including:
- Java-to-WSDL mapping
- WSDL-to-Java mapping
- Mapping between WSDL and SOAP messages

Java-to-WSDL mapping

This section summarizes the Java-to-WSDL mapping rules. The Java-to-WSDL mapping rules are used by the **Java2WSDL** command for *bottom-up processing*. In bottom-up processing, an existing Java service implementation is used to create a WSDL file defining the web service. The generated WSDL file can require additional manual editing for the following reasons:
- Not all Java classes and constructs have mappings to WSDL files. For example, Java classes that do not comply with the Java bean specification rules might not map to a WSDL construct.
- Some Java classes and constructs have multiple mappings to a WSDL file. For example, a java.lang.String class can map to either an xsd:string or soapenc:string construct. The **Java2WSDL** command chooses one of these mappings, but you must edit the WSDL file if a different mapping is required.
- Multiple ways exist to generate WSDL constructs. For example, you can generate the wsdl:part in wsdl:message with a type or element attribute. The **Java2WSDL** command makes an informed choice based on the -style and -use option settings.
- The WSDL file describes the instance data elements sent in the SOAP message. If you want to modify the names or format used in the message, the WSDL file must be edited. For example, the **Java2WSDL** command maps a Java bean property as an XML element. In some circumstances, you might want to change the WSDL file to map the Java bean property as an XML attribute.
- The WSDL file requires editing if header or attachment support is desired.
- The WSDL file requires editing if a multipart WSDL file, using the wsdl:import construct, is desired.

For simple services, the generated WSDL file is sufficient. For complicated services, the generated WSDL file is a good starting point. Read about the Java2WSDL command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

General issues

- Package to namespace mapping:

  The JAX-RPC specification does not indicate the default mapping of Java package names to XML namespaces. The JAX-RPC specification does specify that each Java package must map to a single XML namespace. Likewise, each XML namespace must map to a single Java package. A default mapping algorithm is provided that constructs the namespace by reversing the names of the Java package and adding an `http://` prefix. For example, a package named, com.ibm.webservice, is mapped to the XML namespace `http://webservice.ibm.com`.

  You can override the default mapping between XML namespaces and Java package names by using the -NStoPkg and -PkgtoNS options of the **WSDL2Java** and **Java2WSDL** commands.
- Identifier mapping :

  Java identifiers are mapped directly to WSDL and XML identifiers.

  Java bean property names are mapped to XML identifiers. For example, a Java bean, with getInfo and setInfo methods, maps to an XML construct with the name, `info`.

  The service endpoint interface method parameter names, if available, are mapped directly to the WSDL and XML identifiers. See the information for the WSDL2Java command -implClass option to learn more.
- WSDL construction summary:

Table 133. Mapping of Java to WSDL or an XML construct. Describes the mapping from a Java construct to the related WSDL and XML construct.

Java construct	WSDL and XML construct
Service endpoint interface	wsdl:portType

Table 133. Mapping of Java to WSDL or an XML construct (continued). Describes the mapping from a Java construct to the related WSDL and XML construct.

Java construct	WSDL and XML construct
Method	wsdl:operation
Parameters	wsdl:input, wsdl:message, wsdl:part
Return	wsdl:output, wsdl:message, wsdl:part
Throws	wsdl:fault, wsdl:message, wsdl:part
Primitive types	xsd and soapenc simple types
Java beans	xsd:complexType
Java bean properties	Nested xsd:elements of xsd:complexType
Arrays	JAX-RPC defined xsd:complexType or xsd:element with a maxOccurs="unbounded" attribute
User defined exceptions	`xsd:complexType`

- **Binding** and **service** construction

  A wsdl:binding that conforms to the generated wsdl:portType is generated. A wsdl:service containing a port that references the generated wsdl:binding is generated. The names of the binding and service are controlled by the **Java2WSDL** command.
- **Style** and **use**

  Use the -style and -use options to generate different kinds of WSDL files. The four supported combinations are:
  – -style DOCUMENT -use LITERAL
  – -style RPC -use LITERAL
  – -style DOCUMENT -use LITERAL -wrapped false
  – -style RPC -use ENCODED

  The following is a brief description of each combination.
  – DOCUMENT LITERAL:

    The **Java2WSDL** command generates a Web Services - Interoperability (WS-I) specification compliant document-literal WSDL file. The wsdl:binding is generated with embedded style="document" and use="literal" attributes. An xsd:element is generated for each service endpoint interface method to describe the request message. A similar xsd:element is generated for each service endpoint interface method to describe the response message.
  – RPC LITERAL:

    The **Java2WSDL** command generates a WS-I compliant rpc-literal WSDL file. The wsdl:binding is generated with embedded style="rpc" and use="literal" attributes. The wsdl:message constructs are generated for the inputs and outputs of each service endpoint interface method. The parameters of the method are described by the part elements within the wsdl:message constructs.
  – DOCUMENT LITERAL not wrapped:

    The **Java2WSDL** command generates a document-literal WSDL file according to the JAX-RPC specification. This WSDL file is not compliant with .NET. The main difference between DOCUMENT LITERAL and DOCUMENT LITERAL not wrapped is the use of wsdl:message constructs to define the request and response messages.
  – RPC ENCODED:

    The **Java2WSDL** command generates an rpc-encoded WSDL file according to the JAX-RPC specification. This WSDL file is not compliant with the WS-I specification. The wsdl:binding is generated with embedded style="rpc" and use="encoded" attributes. Certain soapenc mappings are used to represent types and arrays.

  Many Java types map directly to standard XML types. For example, a java.lang.String maps to an xsd:string. These mappings are described in the JAX-RPC specification.

  Java types that cannot be mapped directly to standard XML types are generated in the `wsdl:types` section. A Java class that matches the Java bean pattern is mapped to an `xsd:complexType`. Review the JAX-RPC specification for a description of all the mapping rules. The following example illustrates the mapping for a sample base and derived Java classes.

Java:

```
public abstract class Base {
     public Base() {}
     public int a;                          // mapped
     private int b;                         // mapped via setter/getter
     private int c;                         // not mapped
     private int[] d;                       // mapped via indexed setter/getter

     public int getB() { return b;}         // map property b
     public void setB(int b) {this.b = b;}

     public int[] getD() { return d;}       // map indexed property d
     public void setD(int[] d) {this.d = d;}
     public int getD(int index) { return d[index];}
     public void setB(int index, int value) {this.d[index] = value;}

     public void someMethod() {...}         // not mapped
  }

  public class Derived extends Base {
     public int x;                          // mapped
     private int y;                         // not mapped
  }
```

Mapped to:

```
<xsd:complexType name="Base" abstract="true">
 <xsd:sequence>
  <xsd:element name="a" type="xsd:int"/>
  <xsd:element name="b" type="xsd:int"/>
  <xsd:element name="d" minOccurs="0" maxOccurs="unbounded" type="xsd:int"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
 <xsd:complexContent>
  <xsd:extension base="ns:Base">
   <xsd:sequence>
    <xsd:element name="x" type="xsd:int"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

- Unsupported classes:

  If a class cannot be mapped to an XML type, the **Java2WSDL** command issues a message and an xsd:anyType reference is generated in the WSDL file. In these situations, modify the web service implementation to use the JAX-RPC compliant classes.

WSDL-to-Java mapping

The **WSDL2Java** command generates Java classes using information described in the WSDL file.

General issues:

- Mapping of a namespace to a package:

  JAX-RPC does not specify the mapping of XML namespaces to Java package names. JAX-RPC does specify that each Java package map to a single XML namespace. Likewise, each XML namespace must map to a single Java package. A default mapping algorithm omits any protocol from the XML namespace and reverses the names. For example, an XML namespace `http://websphere.ibm.com` becomes a Java package with the name com.ibm.websphere.

  The default mapping of an XML namespace to a Java package disregards the context-root. If two namespaces are the same up to the first slash, they map to the same Java package. For example, the XML namespaces `http://websphere.ibm.com/foo` and `http://websphere.ibm.com/bar` map to the com.ibm.websphere Java package. You can override the default mapping between XML namespaces and Java package names by using the -NStoPkg and -PkgtoNS options of the **WSDL2Java** and **Java2WSDL** commands.

  XML names are much richer than Java identifiers. They can include characters that are not permitted in Java identifiers. See Appendix 20 of the JAX-RPC specification for the rules to map an XML name to a Java identifier.

- Java construction summary:

The following table summarizes the XML to Java construction. See the JAX-RPC specification for a description of these mappings.

Table 134. Mapping of a WSDL or an XML construct to Java. Describes the mapping between constructions for XML and Java.

WSDL and XML construction	Java construction
xsd:complexType	Java bean class, Java exception class, or Java array
nested xsd:element/xsd:attribute	Java bean property
xsd:simpleType (enumeration)	JAX-RPC enumeration class
wsdl:message The method parameter signature typically is determined by the wsdl:message.	Service endpoint interface method signature
wsdl:portType	Service endpoint interface
wsdl:operation	Service endpoint interface method
wsdl:binding	Stub
wsdl:service	Service interface
wsdl:port	Port accessor method in Service interface

- Mapping standard XML types:
  - JAX-RPC simple XML types mapping:

    Many XML types are mapped directly to Java types. See the JAX-RPC specification for a description of these mappings.

    The **WSDL2Java** command generates Java types for the XML schema constructs that are defined in the wsdl:types section. The XML schema language is broader than the required or optional subset defined in the JAX-RPC specification. The **WSDL2Java** command supports the required mappings and most of the optional mappings, as well as some XML schema mappings that are not included in the JAX-RPC specification. The **WSDL2Java** command ignores some constructs that it does not support. For example, the command does not support the default attribute. If an xsd:element is defined with the default attribute, the default attribute is ignored. In some cases, the command maps unsupported constructs to the Java interface, javax.xml.soap.SOAPElement.

    The standard Java bean mapping is defined in section 4.2.3 of the JAX-RPC specification. The xsd:complexType defines the type. The nested xsd:elements within the xsd:sequence or xsd:all groups are mapped to Java bean properties. For example:

XML:

```
<xsd:complexType name="Sample">
  <xsd:sequence>
  <xsd:element name="a" type="xsd:string"/>
  <xsd:element name="b" maxOccurs="unbounded" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Java:

```
public class Sample {
     // ..
     public Sample() {}

     // Bean Property a
     public String getA()             {...}
     public void   setA(String value) {...}

     // Indexed Bean Property b
     public String[] getB()           {...}
     public String   getB(int index)  {...}
     public void     setB(String[] values) {...}
     public void     setB(int index, String value) {...}

  }
```

  - Mapping of the wsdl:portType construct:

    The wsdl:portType construct is mapped to the service endpoint interface. The name of the wsdl:portType construct is mapped to the name of the class of the service endpoint interface.
  - Mapping of the wsdl:operation construct:

A wsdl:operation construct within a wsdl:portType is mapped to a method of the service endpoint interface. The name of the wsdl:operation is mapped to the name of the method. The wsdl:operation contains wsdl:input and wsdl:output elements that reference the request and response wsdl:message constructs using the message attribute. The wsdl:operation can contain a wsdl:fault element that references a wsdl:message describing the fault. These faults are mapped to Java classes that extend the exception, java.lang.Exception as discussed in section 4.3.6 of the JAX-RPC specification.

- Effect of document literal wrapped format:

  If the WSDL file uses the document literal wrapped format, the method parameters are mapped from the wrapper xsd:element. The document literal wrapped and literal format is automatically detected by the **WSDL2Java** command. The following criteria must be met:
  - The WSDL file must have style="document" in its wsdl:binding construct.
  - The input and output constructs of the operations within the wsdl:binding must contain soap:body elements that contain use="literal".
  - The wsdl:message referenced by the wsdl:operation input construct must have a single part.
  - The part must use the element attribute to reference an xsd:element.
  - The referenced xsd:element, or wrapper element, must have the same name as the wsdl:operation.
  - The wrapper element must not contain any xsd:attributes.

  In such cases, each parameter name is mapped from a nested xsd:element contained within wrapper element. The type of the parameter is mapped from the type of the nested xsd:element. For example:

WSDL:

```
<xsd:element name="myMethod">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="param1" type="xsd:string"/>
   <xsd:element name="param2" type="xsd:int"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
...
<wsdl:message name="response"/>
 <part name="parameters" element="ns:myMethod"/>
</wsdl:message name="response"/>

<wsdl:message name="response"/>
...
<wsdl:operation name="myMethod">
 <input name="input" message="request"/>
 <output name="output" message="response"/>
</wsdl:operation>
```

Java:

```
void myMethod(String param1, int param2) ...
```

- Parameter mapping:

  If the document and literal wrapped format is not detected, the parameter mapping follows the normal JAX-RPC mapping rules set in section 4.3.4 of the JAX-RPC specification.

  Each parameter is defined by a wsdl:message part referenced from the input and output elements.
  - A wsdl:part in the request wsdl:message is mapped to an input parameter.
  - A wsdl:part in the response wsdl:message is mapped to the return value. If multiple wsdl:parts exist in the response message, they are mapped to output parameters.
    - A Holder class is generated for each output parameter, as discussed in section 4.3.5 of the JAX-RPC specification.
  - A wsdl:part that is both the request and response wsdl:message is mapped to an inout parameter.
    - A Holder class is generated for each inout parameter, as discussed in section 4.3.5 of the JAX-RPC specification.
    - The wsdl:operation parameterOrder attribute defines the order of the parameters.

XML:

```
<wsdl:message name="request">
 <part name="param1" type="xsd:string"/>
```

```
 <part name="param2" type="xsd:int"/>
</wsdl:message name="response"/>

<wsdl:message name="response"/>
...
<wsdl:operation name="myMethod" parameterOrder="param1, param2">
 <input name="input" message="request"/>
 <output name="output" message="response"/>
</wsdl:operation>
```

Java:

```
void myMethod(String param1, int param2) ...
```

– Mapping of wsdl:binding:

The **WSDL2Java** command uses the wsdl:binding information to generate an implementation-specific client-side stub. WebSphere Application Server uses the wsdl:binding information on the server side to properly deserialize the request, invoke the web service, and serialize the response. The information in the wsdl:binding does not affect the generation of the service endpoint interface, except when the document and literal wrapped format is used, or when MIME attachments are present.

- MIME attachments:

  For a WSDL 1.1-compliant WSDL file, the part of an operation message, that is defined in the binding as a MIME attachment, becomes a parameter of the type of the attachment regardless of the part declared. For example:

XML:

```
<wsdl:types>
 <schema ...>
  <complexType name="ArrayOfBinary">
   <restriction base="soapenc:Array">
     <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:binary[]"/>
   </restriction>
  </complexType>
 </schema>
</wsdl:types>

<wsdl:message name="request">
 <part name="param1" type="ns:ArrayOfBinary"/>
<wsdl:message name="response"/>

<wsdl:message name="response"/>
 ...

 <wsdl:operation name="myMethod">
  <input name="input" message="request"/>
  <output name="output" message="response"/>
 </wsdl:operation>
  ...

<binding ...
 <wsdl:operation name="myMethod">
  <input>
   <mime:multipartRelated>
    <mime:part>
     <mime:content part="param1" type="image/jpeg"/>
    </mime:part>
   </mime:multipartRelated>
  </input>
   ...
 </wsdl:operation>
```

Java:

```
void myMethod(java.awt.Image param1) ...
```

The JAX-RPC specification requires support for the following MIME types:

Table 135. Mapping of MIME type and Java type. Describes the mapping between MIME types and Java types.

MIME type	Java type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
multipart/*	javax.mail.internet.MimeMultipart
text/xml	javax.xml.transform.Source

Table 135. Mapping of MIME type and Java type (continued). Describes the mapping between MIME types and Java types.

MIME type	Java type
application/xml	javax.xml.transform.Source

– Mapping of wsdl:service:

  The wsdl:service element is mapped to a generated service interface. The generated service interface contains methods to access each of the ports in the wsdl:service element. The generated service interface is discussed in sections 4.3.9, 4.3.10, and 4.3.11 of the JAX-RPC specification.

  In addition, the wsdl:service element is mapped to the implementation-specific ServiceLocator class, which is an implementation of the generated service interface.

Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Mapping between WSDL and SOAP messages

The WSDL file defines the format of the SOAP message that are transmitted through network connections. The **WSDL2Java** command and the WebSphere Application Server runtime use the information in the WSDL file to ensure that the SOAP message is properly serialized and deserialized.

If a wsdl:binding element indicates that a message is sent using an RPC format, the SOAP message contains an element defining the operation. If a wsdl:binding element indicates that the message is sent using a document format, the SOAP message does not contain the operation element.

If the wsdl:part element is defined using the type attribute, the name and type of the part are used in the message. If the wsdl:part element is defined using the element attribute, the name and type of the element are used in the message. The element attribute is not supported by the JAX-RPC specification when use="encoded".

If a wsdl:binding element indicates that a message is encoded, the values in the message are sent with xsi:type information. If a wsdl:binding element indicates that a message is literal, the values in the message are typically not sent with xsi:type information. For example:

DOCUMENT/LITERAL
WSDL:

```
<xsd:element name="c" type="xsd:int"/>
<xsd:element name="method">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="a" type="xsd:string"/>
   <xsd:element ref="ns:c"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
...
  <wsdl:message name="request">
    <part name="parameters" element="ns:method"/>
  </wsdl:message>
 ...
 <wsdl:operation name="method">
  <input message="request"/>
 ...
```

Message:

```
<soap:body>
  <ns:method>
   <a>ABC</a>
   <c>123</a>
  <ns:method>
</soap:body>
```

RPC/ENCODED
WSDL:

```
<xsd:element name="c" type="xsd:int"/>

...
  <wsdl:message name="request">
    <part name="a" type="xsd:string"/>
```

```
    <part name="b" element="ns:c"/>
  </wsdl:message>
 ...
 <wsdl:operation name="method">
    <input message="request"/>
 ...

Message:
<soap:body>
  <ns:method>
   <a xsi:type="xsd:string">ABC</a>
   <element attribute is not permitted in rpc/encoded mode>
  </ns:method>
 </soap:body>
DOCUMENT/LITERAL  not wrapped
WSDL:
<xsd:element name="c" type="xsd:int"/>

...
  <wsdl:message name="request">
    <part name="a" type="xsd:string"/>
    <part name="b" element="ns:c"/>
  </wsdl:message>
 ...
 <wsdl:operation name="method">
    <input message="request"/>

...

Message:
<soap:body>
  <a>ABC</a>
  <c>123</a>
</soap:body>
```

Developing JAX-RPC web services deployment descriptor templates for a JavaBeans implementation

Deployment descriptors are standard text files, formatted using XML and packaged in a web services application. Deployment descriptors are required to deploy Java API for XML-based RPC (JAX-RPC) web services that are developed using Web Services for Java Platform, Enterprise Edition (Java EE) technology.

Before you begin

Develop a Web Services Description Language (WSDL) file.

You need a WSDL file to use web services. You can develop your own WSDL file or get one from a web services provider through email, downloading, or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own. See the developing a WSDL file for JAX-RPC applications information.

About this task

Completing this task creates the deployment descriptors used to describe how to map the service implementation to a JavaBeans component for Java API for XML-based RPC (JAX-RPC) applications.

To develop the deployment descriptor templates from a WSDL file, you must obtain the web address of the WSDL file.

If the WSDL file is a local file and you are running on the Windows platform, the web address looks like this example: `file:drive:\path\file_name.wsdl`. If you are using the Linux or Unix platform, the Web address looks like this example: `file:/path/file_name.wsdl`. You can also specify local files using the absolute or relative file system path.

When the web service is a JavaBeans implementation in a web module, the `webservices.xml`,`ibm-webservices-bnd.xmi` and `ibm-webservices.ext.xmi` deployment descriptors and the JAX-RPC mapping file are generated in the `WEB-INF` subdirectory.

Procedure

Run the **WSDL2Java -verbose -role develop-server -container web -genJava no *wsdlURL*** command to generate the server deployment descriptor templates and mapping file into the `WEB-INF` subdirectory. If the **-verbose** option is specified, a list of all the generated files is displayed when the command runs.

Note: The **WSDL2Java** command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Results

You have deployment descriptor templates that are required to implement or use JAX-RPC web services.

Example

The following example uses a WSDL file named `AddressBookJ2WB.wsdl`:

Generate the template files:

```
WSDL2Java -verbose -role develop-server -container web -genJava no AddressBookJ2WB.wsdl
```

The deployment descriptor templates and mapping file are generated into the `WEB-INF` subdirectory:

```
Parsing XML file: AddressBookJ2WB.wsdl
Generating: WEB-INF\webservices.xml
Generating: WEB-INF\ibm-webservices-bnd.xmi
Generating: WEB-INF\ibm-webservices-ext.xmi
Generating: WEB-INF\AddressBookJ2WB_mapping.xml
```

What to do next

Now, you need to configure the `webservices.xml` deployment descriptor and configure the `ibm-webservices-bnd.xmi` deployment descriptor so that application server can process the incoming web services. After you configure the deployment descriptors, you must assemble the web services application for deployment. See the information on configuring the `webservices.xml` deployment descriptor for JAX-RPC web services and configuring the `ibm-webservices-bnd.xmi` deployment descriptor for JAX-RPC web services.

WSDL2Java command for JAX-RPC applications:

Run the **WSDL2Java** command-line tool against the WSDL file to create Java APIs and deployment descriptor templates.

The WSDL2Java command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform.

A Web Services Description Language (WSDL) file describes a Web service. The Java API for XML-based Remote Procedure Call (JAX-RPC) 1.1 specification defines a Java API mapping that interacts with the web service. The Web Services for Java Platform, Enterprise Edition (Java EE) specification defines deployment descriptors that deploy a web service in a Java EE environment. The **WSDL2Java** command is run against the WSDL file to create Java APIs and deployment descriptor templates according to these specifications.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the **WSDL2Java** tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. However,

if you have an established WSDL or schema that relies on relative namespaces, under specific conditions you can use the `allowRelativeNamespace` property to disable the relative namespace restrictions in the **WSDL2Java** tool. For additional information, see the property description.

You can convert any relative namespaces to absolute namespaces. The following is an example of a relative namespace:

```
targetNamespace="MyRelNamespace"
```

. You can change the relative namespace in this example to an absolute namespace by adding the protocol and base URI information:

```
targetNamespace="http://www.sample.com/MyRelNamespace"
```

Command-line syntax

The command-line syntax is:

```
WSDL2Java [arguments] WSDL-URI
```

Required arguments

- WSDL-URI

  Specifies the location of the input WSDL file using a Universal Resource Identifier (URI). You can also use a regular file path if the WSDL file is on the local file system.

Important arguments

- -role *Java EE role*

  Specifies the Java EE development role that identifies which files to generate. Valid arguments include:
  - client

    A combination of the develop-client and deploy-client arguments.
  - deploy-client

    Generates binding files for client deployment.
  - deploy-server

    Generates binding files for server deployment.
  - develop-client (default)

    Generates files for client development.
  - develop-server

    Generates files for server development.
  - server

    A combination of the develop-server and deploy-server arguments.
- -container *Java EE-container*

  Indicates the Java EE container to use. Valid arguments include:
  - client

    Indicates client container.
  - ejb

    Indicates an Enterprise JavaBeans (EJB) container.
  - none

    Indicates no container.
  - web

    Indicates a web container.

  For client roles (see the -role option), the default argument is `none`. For server roles, the container must be `ejb` or `web`. The same container option must be used for both development and deployment.
- -output *directory*

  Sets the root directory for emitted files.
- -inputMappingFile mapping file

Specifies the file name of the Web Services for Java EE mapping file.

- -introspect

  Uses existing Java beans with a new web service API.

  In some scenarios, it is good to use existing Java classes instead of generating new classes. The -introspect option directs the **WSDL2Java** command to examine existing Java classes when generating classes. The existing classes are validated against the JAX-RPC specification. For example:

  Suppose you have an existing Java bean

```
public class Bean {
 public Date x;
}
```

  The WSDL file defines *x* as `xsd:dateTime`. Without the -introspect option, the **WSDL2Java** command generates a Java bean that is similar to the following example:

```
public class Bean {
 private Calendar x;
 public void setx(Calendar value) (x=value;)
 public Calendar getX() { return x;)
}
```

  The **WSDL2Java** command uses the -introspect option to examine the original Java bean and to generate classes that are compatible with existing Java beans.

- -classpath *paths*

  Defines an alternative class path to search for Java classes.

- -noDataBinding

  Disables the binding of XML types to Java types. Instead, each XML type is mapped to a `javax.xml.soap.SOAPElement` interface defined by the SOAP with Attachments API for Java (SAAJ) specification.

  The Java API for XML Web Services (JAX-WS) programming model supports SAAJ 1.2 and 1.3.

  The JAX-RPC programming model supports SAAJ 1.2.

  The Java programming models define Java mappings for a subset of XML types. Several XML types cannot be mapped to Java beans or primitives. In this situation, the **WSDL2Java** command maps the type to an SAAJ `SOAPElement`. A SAAJ SOAPElement is a generic representation of the element in the message. The methods on the `SOAPElement` can be used to examine the element and its children.

  In some scenarios, it might be more appropriate to use the generic `SOAPElement` mapping exclusively. To learn more about the use of `SOAPElement`, see the information on SOAP with Attachments API for Java interface and custom data binders for JAX-RPC applications.

  For a complete list of the supported standards and specifications, see the web services specifications and API documentation.

- -help

  Displays a help message and exits.

- -helpX

  Displays a help message for extended options. The options include:

- -verbose

  Displays processing information, including the names of the generated files.

- -NStoPkg *namespace=package*

  By default, package names are automatically derived from the namespace strings in the WSDL file. For example, if the namespace is of the form *http://x.y.com* or *urn:x.y.com*, the corresponding package is *com.y.x*.

  You can provide your own mapping by using the -NStoPkg argument, which you can repeat as often as necessary, once for each unique namespace mapping. For example, if a namespace in the WSDL file is called `urn:AddressFetcher2`, and you want files generated from the objects in this namespace to reside in the samples.addr package, provide the -NStoPkg "http://urn:AddressFetcher2/"=samples.addr argument to the **WSDL2Java** command.

- -timeout *seconds*

Specifies how long the **WSDL2Java** command waits, in seconds, for the WSDL-URI to respond before giving up. The default is 45 seconds; -1 disables the timeout.

- -genResolver

  Generates an absolute-import resolver class. The purpose of this class is to record the contents of the imported WSDL files that are used by the WSDL URI. This class is used by the run time and can also be used for future **WSDL2Java** command runs. This flexibility is desirable when the imported WSDL files are remote and possibly inaccessible. When an import resolver is used, the possibility that a remote WSDL file has different contents at run time that it did during development is eliminated. The generated class is named `_AbsoluteImportResolver.java`. Compile and package this class with the other Java classes that are generated by the **WSDL2Java** command.
- -useResolver *resolver-class*

  Specifies an absolute-import resolver class to use during parsing. This class must be created during a previous run of the **WSDL2Java** command that uses the -genResolver option. The class must be available in the CLASSPATH variable.
- -deployScope *argument*

  Indicates how to deploy the server implementation. Valid arguments include:
  - Application

    Uses one instance of the implementation class for all requests.
  - Request

    Creates a new instance of the implementation class for each request.
  - Session

    Creates a new instance of the implementation class for each session.

Other arguments

- -user *id*

  Specifies the login user name to access the WSDL URI.
- -password *password*

  Specifies the login user password to access the WSDL URI.
- -all

  Generates Java files for all types, even those that are not referenced.
- -allowRelativeNamespace *true or false*

  Specifies whether to disable the relative namespace restrictions. If you specify `-allowRelativeNamespace=true`, the relative namespace restrictions are disabled.

  Note: Only use this property if you have an established WSDL file or schema that relies on a relative namespaces and you are seeking to interoperate with a defined set of vendors that permit the use of relative namespaces.
- -debug

  Prints debugging information.
- -genJava *argument*

  Generates Java files. Valid arguments include:
  - IfNotExists, default
  - Overwrite
  - No
- -javaSearch *argument*

  The -javaSearch option is used with the -genJava option. If the -genJava IfNotExists, use the -javaSearch option to determine how file existence is detected.
  - File (default): Looks for a file in the output directory
  - Classpath: Looks for a class in the CLASSPATH variable
  - Both: Looks for a file in the output directory or in a class in the CLASSPATH variable

- -genXML *argument*

  Generates the `.xml` and `.xmi` files. Valid arguments are:
  - IfNotExists, default
  - Overwrite
  - No
- -genImplSer *true or false*

  Indicates that each generated Java bean implements the java.io.Serializable. The default is `false`.
- -genEquals *true or false*

  Indicates that each generated Java bean have `equals` and `hashCode` methods. The default is false.
- -noWrappedOperations

  Disables wrapped operations detection. Java beans for the request and response messages are generated.
- -noWrappedArrays

  Disables wrapped array detection.
- -fileNStoPkg *file name*

  Specifies the file of the namespace to package mappings. The default is NStoPKG.properties.
- service *wsdl service name*

  Generates files for the installed WSDL service only.
- -testCase

  Generates the template for a JUnit test case for testing web services. JUnit is a simple framework to write repeatable tests.

Developing JAX-RPC web services deployment descriptor templates for an enterprise bean implementation

You can develop deployment descriptor templates for an Enterprise JavaBeans (EJB) implementation that is enabled for Java API for XML-based RPC (JAX-RPC) web services.

Before you begin

You need to create a service endpoint interface and develop a Web Services Description Language (WSDL) file before you can develop the deployment descriptor templates because the service endpoint interface and the WSDL file are artifacts that are used to create the templates.

About this task

Completing this task creates deployment descriptor templates that describe how to map the service implementation to a Enterprise JavaBeans (EJB). This task is a required step in developing a web service from an enterprise bean.

To develop the deployment descriptor templates from a WSDL file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

When the web service implementation contains an enterprise bean in an EJB module, the `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file are generated in the `META-INF` subdirectory.

Procedure

Run the `WSDL2Java -verbose -role develop-server -container ejb -genJava no` *wsdlURL* command to generate the server deployment descriptor templates and mapping file into the `META-INF` subdirectory. If the -verbose option is specified, a list of all generated files displays when the command runs.

Note: The **WSDL2Java** command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly provided with the z/OS version of the product. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Results

You have deployment descriptor templates that are required to implement a web service.

Example

The following example uses the `AddressBookJ2WE.wsdl` WSDL file:
1. Generate the template files with the following command syntax:

```
WSDL2Java -verbose -role develop-server -container ejb -genJava no AddressBookJ2WE.wsdl
```

The deployment descriptor templates are generated into the `META-INF` subdirectory as follows:

```
Parsing XML file: AddressBookJ2WE.wsdl
Generating: META-INF\webservices.xml
Generating: META-INF\ibm-webservices-bnd.xmi
Generating: META-INF\ibm-webservices-ext.xmi
Generating: META-INF\AddressBookJ2WE_mapping.xml
```

What to do next

Continue to complete the steps that are necessary to develop a JAX-RPC web service from an enterprise bean. The next step is to complete the EJB implementation. When you complete the EJB implementation, you assemble an enterprise bean Java archive (JAR) file that contains the enterprise bean and supporting classes created from a WSDL file. To learn more, see the completing the EJB implementation for JAX-RPC applications information.

Completing the JavaBeans implementation for JAX-RPC applications

After you have developed the Java artifacts necessary to develop a Java API for XML-based RPC (JAX-RPC) web service, you must complete the JavaBeans implementation to assemble a Java archive (JAR) file or a web application archive (WAR) file based on your programming model. The resulting JAR file or WAR file contains the JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Develop web services deployment descriptor templates for a JavaBeans implementation using the **WSDL2java** command-line tool. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans implementation.

About this task

For JAX-RPC applications, complete the JavaBeans implementation by writing your business application.

Procedure

1. Edit the JavaBeans implementation template, *binding*Impl.java. The *binding* is the name of the <wsdl:binding> element in the WSDL file. The JavaBeans implementation is generated by the **WSDL2java** command-line tool.
   a. Complete the implementation of the methods in the template.
   b. (Optional) Make changes if necessary.
   c. (Optional) Change the class name if the binding name is not acceptable.
2. Compile all the Java classes.
3. Assemble a Web archive (WAR) file. Assemble all the Java classes into a WAR file using web module assembly tools. Include all of the classes generated from running the **WSDL2java** command tool for JAX-RPC web service applications when developing implementation templates and bindings from a WSDL file.

Results

You have now enabled the JavaBeans-based business application for JAX-RPC web services. You have a JAR file or a WAR file containing the JavaBeans implementation and supported classes created from the WSDL file.

What to do next

If you are developing a JAX-RPC web services application from JavaBeans, you need to configure the webservices.xml deployment descriptor and configure the ibm-webservices-bnd.xmi deployment descriptor so that the application server can process the incoming web services requests.

Completing the EJB implementation for JAX-RPC applications

After you have developed the Java artifacts necessary to develop a Java API for XML-based RPC (JAX-RPC) web service, you must complete the Enterprise JavaBeans (EJB) implementation to assemble a Java archive (JAR) file or a web application archive (WAR) file based on your programming model. The resulting JAR file or WAR file contains the Enterprise JavaBeans (EJB) implementation and the supported classes created from the tooling.

Before you begin

Develop EJB implementation templates and bindings from a WSDL file for JAX-RPC web services using the **wsdl2java** command-line tool. The deployment descriptor templates that are generated from a Web Services Description Language (WSDL) file are required to complete the EJB implementation in the web services development process.

About this task

For JAX-RPC applications, complete the enterprise beans implementation by writing your business application.

Procedure

1. Inspect the EJB remote interface template, *portType*_RI.java. If necessary, modify the template. The value *portType* is the name of the <wsdl:portType> element in the WSDL file.
2. Edit the *binding*Impl.java EJB implementation template. Where *binding* is the name of the <wsdl:binding> element in the WSDL file.
3. Complete the implementation of the methods in the template.
4. (Optional) Make changes if necessary.
5. (Optional) Change the class name if the binding name is not acceptable.

6. Compile all the Java classes.
7. Assemble an EJB Java archive (JAR) file. Assemble all the Java classes into an enterprise bean JAR file using assembly tools. Include all of the classes generated from running the **WSDL2Java** command tool when developing implementation templates and bindings from a WSDL file.

Results

You have enabled an enterprise beans business application for JAX-RPC web services. You now have an enterprise bean JAR file containing an EJB and supporting classes created from web services artifacts.

What to do next

Now that you have gathered the required artifacts for developing a JAX-RPC web service with an enterprise bean, you need to, configure the `webservices.xml` deployment descriptor.

Configuring the webservices.xml deployment descriptor for JAX-RPC web services

You can configure the `webservices.xml` deployment descriptor with an assembly tool.

Before you begin

To configure the client deployment descriptor, read about the configuring the client deployment descriptor in the Rational Application Developer information.

Before you can configure the `ibm-webservices-bnd.xml` deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

About this task

For JAX-RPC web services, this task is one of the required steps in developing a web service. You need to configure the deployment descriptors so that the application server can process the incoming web services requests.

If you are developing a web service from JavaBeans, you can develop web services JavaBeans deployment descriptor templates from a Web Services Description Language (WSDL) file. Then, you complete the JavaBeans implementation. To learn more, read about developing JavaBeans deployment descriptor templates from a WSDL file and completing the JavaBeans implementation.

If you are developing a web service from an enterprise bean, you can develop web services Enterprise JavaBeans (EJB) deployment descriptor templates from a WSDL file. Then, you complete the EJB implementation. To learn more, read about developing web services EJB deployment descriptor templates from a WSDL file and completing the EJB implementation.

When the JavaBeans implementation is complete, the web module web application archive (WAR) file is assembled. When the EJB implementation is complete, the enterprise bean Java archive (JAR) file is assembled. These archive files contain the `webservices.xml` deployment descriptor. The archive files must be assembled before you can configure the `webservices.xml` deployment descriptor.

The assembly tools provide a graphical interface for developing code artifacts, assembling the code artifacts into various archives (modules) and configuring compliant deployment descriptors for Java Platform, Enterprise Edition (Java EE).

Configure the `webservices.xml` deployment descriptor by following the steps provided in this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.
4. Configure the deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Results

You have a `webservices.xml` deployment descriptor that is configured.

What to do next

For JAX-RPC web services, you must configure the `ibm-webservices-bnd.xmi` deployment descriptor. To learn more, see the configuring the `ibm-webservices-bnd.xmi` deployment descriptor for JAX-RPC web services information.

Configuring the webservices.xml deployment descriptor for handler classes

You can use an assembly tool to configure the `webservices.xml` deployment descriptor for user-provided handler classes.

Before you begin

You can configure deployment descriptors with assembly tools provided with the application server.

A *handler class* is a class that is written to modify a SOAP message that represents a remote procedure call (RPC) request or response. Handlers can be associated with a web service or a web service client.

Similar to Java API for XML-based RPC (JAX-RPC) web services, you can use deployment descriptors to describe Java API for XML Web Services (JAX-WS) web services. For JAX-WS web services, the use of the `webservices.xml` deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

To complete this task, you need an enterprise archive (EAR) file for the applications that you want to configure. For some handler use, such as logging or tracing, only the server or client application require configuration. For other handler use, including sending information in the SOAP headers, the client and server applications must be configured with symmetrical handlers.

About this task

The modules in the EAR file contain the handler classes to configure. These classes implement the `javax.xml.rpc.handler.Handler` interface. For more information on writing handler classes, see chapter 6 of the Web Services for Java EE specification. See chapter 9 in the JAX-WS specification or chapter 12 in the JAX-RPC specification for additional information on the handler framework for your programming

model. The application modules must contain the webservices.xml deployment descriptor. See the web services specifications and API information to review the JAX-RPC specification along with a complete list of the supported standards and specifications.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.
4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Configuring the ibm-webservices-bnd.xmi deployment descriptor for JAX-RPC web services

Use assembly tools to configure the ibm-webservices-bnd.xml deployment descriptor. This file stores binding information that is associated with the endpoints defined with the webservices.xml deployment descriptor file.

Before you begin

Note: The ibm-webservices-bnd.xmi deployment descriptor is for Java API for XML-based RPC (JAX-RPC) based web services application. It is not used for Java API for XML-Based Web Services (JAX-WS) enabled applications.

To configure the client deployment descriptor, read about the configuring the client deployment descriptor in the Rational Application Developer information.

Before you can configure the ibm-webservices-bnd.xml deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

About this task

This task is one of the steps in developing a web service. You need to configure the deployment descriptors so that WebSphere Application Server can process the incoming web services requests.

Depending on if you are developing a web service from JavaBeans or an enterprise bean:
- Develop web services JavaBeans deployment descriptor templates from a Web Services Description Language (WSDL) file.
- Develop web services Enterprise JavaBeans (EJB) deployment descriptor templates from a WSDL file.

Then, complete the EJB implementation or complete the JavaBeans implementation. When the EJB implementation is complete, the enterprise bean Java archive (JAR) file is assembled. When the JavaBeans implementation is complete, the web module web application archive (WAR) file is assembled. These archive files contain the webservices.xml deployment descriptor. The archive files must be assembled before you can configure the webservices.xml deployment descriptor.

Configure the webservices.xml deployment descriptor by following the steps provided in this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.
4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Results

The `ibm-webservices-bnd.xmi` deployment descriptor is configured for the web service implementation module.

What to do next

If you are developing a web service from JavaBeans, assemble a WAR file that is enabled for web services from Java code. See the assembling a WAR file that is enabled for web services from Java code information.

If you are developing a web service from an enterprise bean, assemble a JAR file that is enabled for web services from an enterprise bean. To learn more about assembling the artifacts that are required to enable the EJB module for web services into the JAR file, see the assembling a JAR file that is enabled for web services from an enterprise bean information.

JAX-RPC web services enabled module - deployment descriptor settings (ibm-webservices-bnd.xmi file)

The `ibm-webservices-bnd.xmi` file is a deployment descriptor for a Java API for XML-based RPC (JAX-RPC) web services-enabled web module or an Enterprise JavaBeans (EJB) module. This file contains information for the web services run time that is required by WebSphere Application Server..

You can edit these properties using an assembly tool. See Configuring the `ibm-webservices-bnd.xmi` deployment descriptor for JAX-RPC web services for instructions.

The following user-defined assembly properties are supported:

- **`wsDescNameLink`**

  Attribute of the wsdescBindings element that specifies the link to the corresponding <webservice-description-name> element in the `webservices.xml` file.
- **pc-name-link**

  Attribute of the pcBindings element that specifies the link to the <port-component-name> element in the `webservices.xml` file.
- **scope**

  Attribute of the pcBindings element that specifies when new instances of implementation beans are created. Possible values are request, session, and application.

You can change scope value for a deployed web service using the administrative console. Click **Enterprise Applications** > *application* > **Web modules** or **EJB modules** > *module* > **Web Services Implementation Scope**.

Bindings file examples

The following examples demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute with a unique string value. Review the template xmi files generated by the **WSDL2Java** command for examples of ID strings. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

```
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0" xmlns:xmi=
"http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
 <wsdescBindings wsDescNameLink="AddressBookService">
  <pcBindings pcNameLink="AddressBook" scope="Application"/>
 </wsdescBindings>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Developing JAX-RPC web services with WSDL files (top-down)

Setting up a development environment for web services

The application server provides command-line tools to develop web services clients and implementations that are based on the Web Services for Java Platform, Enterprise Edition (Java EE) specification. You must set up your development environment before you start developing web services.

Before you begin

Before you can set up a web services development environment within WebSphere Application Server, you must install WebSphere Application Server. For detailed information on installing the application server, read about installing your application server environment.

About this task

Set up a web services development environment by completing the following actions.

Procedure

1. Set up the environment.

   Run the **setupCmdLine** script from the */profile_root/<application_server>/bin* directory.
2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:

```
set PATH=%WAS_PATH%;%PATH%
```

Results

You have set up an environment so that you can develop Web services.

What to do next

Implement web services applications. See the task overview for implementing web services applications information to learn about how to develop and implement a Java EE web service.

Developing Java artifacts for JAX-RPC applications from a WSDL file

You can develop Java artifacts from a Web Services Description Language (WSDL) file for JAX-RPC applications from a WSDL file by using the WSDL2Java command-line tool to create Java implementation templates and bindings.

Before you begin

To develop the JavaBeans implementation templates and bindings from a WSDL file, you must obtain the Uniform Resource Locator (URL) of the WSDL file.

If the WSDL file is a local file, the URL looks like this example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

Implementation templates are generated using the -role develop-server option of the **WSDL2Java** command. The **WSDL2Java** command also generates bindings and deployment descriptors.

The **WSDL2Java** command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the WSDL2Java command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

About this task

Develop JavaBeans implementation templates and bindings from a WSDL file by issuing the proper command.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the **WSDL2Java** tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. You can convert any relative namespaces to absolute namespaces. To learn more about the use of namespaces with the **WSDL2Java** tool, see the **WSDL2Java** command for JAX-RPC applications documentation.

Procedure

Run the `WSDL2Java -verbose -role develop-server -container web` *wsdlURL* command. Since the -verbose option is specified, a list of all the generated files is displayed when the command runs.

Results

You have templates for the implementation and deployment descriptors required to implement a web service, as well as bindings files. These templates are partially filled with information from the WSDL file.

Example

The following example uses the AddressBook JavaBeans implementation and the `AddressBook.wsdl` WSDL file. After generating the template files from the `WSDL2Java -verbose -role develop-server -container web AddressBook.wsdl` command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookSoapBindingImpl.java..
WSWS3282I: Info: Generating WEB-INF\webservices.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating WEB-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-ext.xmi.
```

The `AddressBookSOAPBindingImpl.java` file is the template for the implementation bean. It is named after the port in the WSDL file. Generally, this class is renamed to a more meaningful name.

What to do next

Complete the JavaBeans implementation for JAX-RPC applications.

Developing EJB implementation templates and bindings from a WSDL file for JAX-RPC web services

You can develop Enterprise JavaBeans (EJB) implementation deployment descriptor templates and bindings from a Web Services Description Language (WSDL) file for a JAX-RPC application.

Before you begin

To develop EJB implementation templates and bindings from a WSDL file for a Java API for XML-based RPC (JAX-RPC) web service, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If the WSDL file is a local file, the URL looks like the following example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

About this task

This task is one a required step in developing a web service from an enterprise bean.

Implementation templates are generated using the `-role develop-server` option of the **WSDL2Java** command.

Templates are generated for an EJB implementation for the following components:
- enterprise bean
- EJB remote interface
- EJB Home

The **WSDL2Java** command also generates bindings and deployment descriptors.

The **WSDL2Java** command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the **WSDL2Java** tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. You can convert any relative namespaces to absolute namespaces. To learn more about the use of namespaces with the **WSDL2Java** tool, see the **WSDL2Java** command for JAX-RPC applications documentation.

Procedure

Run the `WSDL2Java -verbose -role develop-server -container ejb` *wsdlURL* command. Because the verbose option is specified, a list of all the generated files is displayed when the command runs.

Results

You have templates for the implementation and deployment descriptors required to implement web services, as well as bindings files. These templates are partially completed with information from the WSDL file.

Example

The following example uses the enterprise bean AddressBook enterprise bean and the `AddressBook.wsdl` file. After generating the template files from the `WSDL2Java -verbose -role develop-server -container EJB AddressBook.wsdl` command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookSoapBindingImpl.java.
WSWS3282I: Info: Generating addr\AddressBook_RI.java.
WSWS3282I: Info: Generating addr\AddressBookHome.java.
WSWS3282I: Info: Generating META-INF\webservices.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservices-ext.xmi.
```

What to do next

Complete the EJB implementation. When you complete the EJB implementation, an EJB Java archive (JAR) file that contains an EJB and supporting classes is created from a WSDL file.

Completing the JavaBeans implementation for JAX-RPC applications

After you have developed the Java artifacts necessary to develop a Java API for XML-based RPC (JAX-RPC) web service, you must complete the JavaBeans implementation to assemble a Java archive (JAR) file or a web application archive (WAR) file based on your programming model. The resulting JAR file or WAR file contains the JavaBeans implementation and the supported classes created from the tooling.

Before you begin

Develop web services deployment descriptor templates for a JavaBeans implementation using the **WSDL2java** command-line tool. You need to complete this step to create the deployment descriptor templates that are configured to map the service implementation to the JavaBeans implementation.

About this task

For JAX-RPC applications, complete the JavaBeans implementation by writing your business application.

Procedure

1. Edit the JavaBeans implementation template, *binding*`Impl.java`. The *binding* is the name of the <wsdl:binding> element in the WSDL file. The JavaBeans implementation is generated by the **WSDL2java** command-line tool.
   a. Complete the implementation of the methods in the template.
   b. (Optional) Make changes if necessary.
   c. (Optional) Change the class name if the binding name is not acceptable.
2. Compile all the Java classes.
3. Assemble a Web archive (WAR) file. Assemble all the Java classes into a WAR file using web module assembly tools. Include all of the classes generated from running the **WSDL2java** command tool for JAX-RPC web service applications when developing implementation templates and bindings from a WSDL file.

Results

You have now enabled the JavaBeans-based business application for JAX-RPC web services. You have a JAR file or a WAR file containing the JavaBeans implementation and supported classes created from the WSDL file.

What to do next

If you are developing a JAX-RPC web services application from JavaBeans, you need to configure the `webservices.xml` deployment descriptor and configure the `ibm-webservices-bnd.xmi` deployment descriptor so that the application server can process the incoming web services requests.

Completing the EJB implementation for JAX-RPC applications

After you have developed the Java artifacts necessary to develop a Java API for XML-based RPC (JAX-RPC) web service, you must complete the Enterprise JavaBeans (EJB) implementation to assemble a Java archive (JAR) file or a web application archive (WAR) file based on your programming model. The resulting JAR file or WAR file contains the Enterprise JavaBeans (EJB) implementation and the supported classes created from the tooling.

Before you begin

Develop EJB implementation templates and bindings from a WSDL file for JAX-RPC web services using the **wsdl2java** command-line tool. The deployment descriptor templates that are generated from a Web Services Description Language (WSDL) file are required to complete the EJB implementation in the web services development process.

About this task

For JAX-RPC applications, complete the enterprise beans implementation by writing your business application.

Procedure

1. Inspect the EJB remote interface template, *portType*`_RI.java`. If necessary, modify the template. The value *portType* is the name of the `<wsdl:portType>` element in the WSDL file.
2. Edit the *binding*`Impl.java` EJB implementation template. Where *binding* is the name of the `<wsdl:binding>` element in the WSDL file.
3. Complete the implementation of the methods in the template.
4. (Optional) Make changes if necessary.
5. (Optional) Change the class name if the binding name is not acceptable.
6. Compile all the Java classes.
7. Assemble an EJB Java archive (JAR) file. Assemble all the Java classes into an enterprise bean JAR file using assembly tools. Include all of the classes generated from running the **WSDL2Java** command tool when developing implementation templates and bindings from a WSDL file.

Results

You have enabled an enterprise beans business application for JAX-RPC web services. You now have an enterprise bean JAR file containing an EJB and supporting classes created from web services artifacts.

What to do next

Now that you have gathered the required artifacts for developing a JAX-RPC web service with an enterprise bean, you need to, configure the `webservices.xml` deployment descriptor.

Configuring the webservices.xml deployment descriptor for JAX-RPC web services

You can configure the `webservices.xml` deployment descriptor with an assembly tool.

Before you begin

To configure the client deployment descriptor, read about the configuring the client deployment descriptor in the Rational Application Developer information.

Before you can configure the `ibm-webservices-bnd.xml` deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

About this task

For JAX-RPC web services, this task is one of the required steps in developing a web service. You need to configure the deployment descriptors so that the application server can process the incoming web services requests.

If you are developing a web service from JavaBeans, you can develop web services JavaBeans deployment descriptor templates from a Web Services Description Language (WSDL) file. Then, you complete the JavaBeans implementation. To learn more, read about developing JavaBeans deployment descriptor templates from a WSDL file and completing the JavaBeans implementation.

If you are developing a web service from an enterprise bean, you can develop web services Enterprise JavaBeans (EJB) deployment descriptor templates from a WSDL file. Then, you complete the EJB implementation. To learn more, read about developing web services EJB deployment descriptor templates from a WSDL file and completing the EJB implementation.

When the JavaBeans implementation is complete, the web module web application archive (WAR) file is assembled. When the EJB implementation is complete, the enterprise bean Java archive (JAR) file is assembled. These archive files contain the `webservices.xml` deployment descriptor. The archive files must be assembled before you can configure the `webservices.xml` deployment descriptor.

The assembly tools provide a graphical interface for developing code artifacts, assembling the code artifacts into various archives (modules) and configuring compliant deployment descriptors for Java Platform, Enterprise Edition (Java EE).

Configure the `webservices.xml` deployment descriptor by following the steps provided in this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.
4. Configure the deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Results

You have a webservices.xml deployment descriptor that is configured.

What to do next

For JAX-RPC web services, you must configure the ibm-webservices-bnd.xmi deployment descriptor. To learn more, see the configuring the ibm-webservices-bnd.xmi deployment descriptor for JAX-RPC web services information.

Configuring the webservices.xml deployment descriptor for handler classes

You can use an assembly tool to configure the webservices.xml deployment descriptor for user-provided handler classes.

Before you begin

You can configure deployment descriptors with assembly tools provided with the application server.

A *handler class* is a class that is written to modify a SOAP message that represents a remote procedure call (RPC) request or response. Handlers can be associated with a web service or a web service client.

Similar to Java API for XML-based RPC (JAX-RPC) web services, you can use deployment descriptors to describe Java API for XML Web Services (JAX-WS) web services. For JAX-WS web services, the use of the webservices.xml deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the webservices.xml deployment descriptor overrides any corresponding information that is specified by annotations.

To complete this task, you need an enterprise archive (EAR) file for the applications that you want to configure. For some handler use, such as logging or tracing, only the server or client application require configuration. For other handler use, including sending information in the SOAP headers, the client and server applications must be configured with symmetrical handlers.

About this task

The modules in the EAR file contain the handler classes to configure. These classes implement the javax.xml.rpc.handler.Handler interface. For more information on writing handler classes, see chapter 6 of the Web Services for Java EE specification. See chapter 9 in the JAX-WS specification or chapter 12 in the JAX-RPC specification for additional information on the handler framework for your programming model. The application modules must contain the webservices.xml deployment descriptor. See the web services specifications and API information to review the JAX-RPC specification along with a complete list of the supported standards and specifications.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate

files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.

4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Configuring the ibm-webservices-bnd.xmi deployment descriptor for JAX-RPC web services

Use assembly tools to configure the `ibm-webservices-bnd.xml` deployment descriptor. This file stores binding information that is associated with the endpoints defined with the `webservices.xml` deployment descriptor file.

Before you begin

Note: The `ibm-webservices-bnd.xmi` deployment descriptor is for Java API for XML-based RPC (JAX-RPC) based web services application. It is not used for Java API for XML-Based Web Services (JAX-WS) enabled applications.

To configure the client deployment descriptor, read about the configuring the client deployment descriptor in the Rational Application Developer information.

Before you can configure the `ibm-webservices-bnd.xml` deployment descriptor, you must develop the deployment descriptor templates and complete the implementation.

About this task

This task is one of the steps in developing a web service. You need to configure the deployment descriptors so that WebSphere Application Server can process the incoming web services requests.

Depending on if you are developing a web service from JavaBeans or an enterprise bean:

- Develop web services JavaBeans deployment descriptor templates from a Web Services Description Language (WSDL) file.
- Develop web services Enterprise JavaBeans (EJB) deployment descriptor templates from a WSDL file.

Then, complete the EJB implementation or complete the JavaBeans implementation. When the EJB implementation is complete, the enterprise bean Java archive (JAR) file is assembled. When the JavaBeans implementation is complete, the web module web application archive (WAR) file is assembled. These archive files contain the `webservices.xml` deployment descriptor. The archive files must be assembled before you can configure the `webservices.xml` deployment descriptor.

Configure the `webservices.xml` deployment descriptor by following the steps provided in this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.
4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Results

The ibm-webservices-bnd.xmi deployment descriptor is configured for the web service implementation module.

What to do next

If you are developing a web service from JavaBeans, assemble a WAR file that is enabled for web services from Java code. See the assembling a WAR file that is enabled for web services from Java code information.

If you are developing a web service from an enterprise bean, assemble a JAR file that is enabled for web services from an enterprise bean. To learn more about assembling the artifacts that are required to enable the EJB module for web services into the JAR file, see the assembling a JAR file that is enabled for web services from an enterprise bean information.

JAX-RPC web services enabled module - deployment descriptor settings (ibm-webservices-bnd.xmi file)

The ibm-webservices-bnd.xmi file is a deployment descriptor for a Java API for XML-based RPC (JAX-RPC) web services-enabled web module or an Enterprise JavaBeans (EJB) module. This file contains information for the web services run time that is required by WebSphere Application Server..

You can edit these properties using an assembly tool. See Configuring the ibm-webservices-bnd.xmi deployment descriptor for JAX-RPC web services for instructions.

The following user-defined assembly properties are supported:

- **wsDescNameLink**

  Attribute of the wsdescBindings element that specifies the link to the corresponding <webservice-description-name> element in the webservices.xml file.
- **pc-name-link**

  Attribute of the pcBindings element that specifies the link to the <port-component-name> element in the webservices.xml file.
- **scope**

  Attribute of the pcBindings element that specifies when new instances of implementation beans are created. Possible values are request, session, and application.

You can change scope value for a deployed web service using the administrative console. Click **Enterprise Applications** > *application* > **Web modules** or **EJB modules** > *module* > **Web Services Implementation Scope**.

Bindings file examples

The following examples demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute with a unique string value. Review the template xmi files generated by the **WSDL2Java** command for examples of ID strings. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

```
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0" xmlns:xmi=
"http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wsbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
 <wsdescBindings wsDescNameLink="AddressBookService">
  <pcBindings pcNameLink="AddressBook" scope="Application"/>
 </wsdescBindings>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Developing JAX-RPC web services clients

Developing client bindings from a WSDL file for a JAX-RPC Web services client

You can develop client bindings from a Web Services Description (WSDL) file for a JAX-RPC web services client.

Before you begin

To develop the client bindings from a WSDL file for JAX-RPC web service applications, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use. You need bindings and deployment descriptors in order for a client to use a web service.

If the WSDL file is a local file, the URL looks like the following example: file:*drive:**path**file_name*.wsdl.

You can also specify local files using the absolute or relative file system path.

Client bindings are generated using the `-role develop-client` option in combination with the `-container` option of the **WSDL2Java** command. The `-container` option takes the following parameters:

- **-container client**

  Generates bindings and deployment descriptors for a client residing in the application client container.
- **-container ejb**

  Generates bindings and deployment descriptors for a client that is an enterprise bean in the Enterprise JavaBeans (EJB) module.
- **-container web**

  Generates bindings and deployment descriptors for a client residing in the web container.

The **WSDL2Java** command-line tool is not supported on the z/OS platform. This functionality is provided by the assembly tools provided with the z/OS version of the product. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

About this task

Develop client bindings from a WSDL file by running the appropriate command.

Note: It is a best practice to use absolute namespaces within your WSDL or schema. By default, the **WSDL2Java** tool does not permit the use of relative namespaces. Relative namespaces have been deprecated by the XML Plenary Interest Group and the use of relative namespaces causes the XML Digital Signature to fail as required by the Canonical XML Version 1.0 specification. You can convert any relative namespaces to absolute namespaces. To learn more about the use of namespaces with the **WSDL2Java** tool, see the **WSDL2Java** command for JAX-RPC applications documentation.

Procedure

Run the `WSDL2Java -verbose -role develop-client -container` *type wsdlURL* command, where *type* is **ejb** for an enterprise EJB client, `web` for a JavaBeans client, or `client` for an application client.
You can use the following combinations in the command-line:

- -container `web`
- -container `ejb`
- -container `client`

Because the verbose option is specified, a list of all generated files is displayed when the command runs.

Results

You have the bindings and deployment descriptors needed by a client to use a web service.

Example

The following example uses the AddressBook enterprise bean the `AddressBook.wsdl` WSDL file. After generating the bindings from the `WSDL2Java -verbose -role develop-client -container client AddressBook.wsdl` command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookService.java.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-ext.xmi.
```

What to do next

Complete the client implementation by writing your client application and then assembling the client artifacts.

.

Changing SOAP message encoding to support WSI-Basic Profile

Support for Universal Transformation Format (UTF)-16 encoding is required by the WS-I Basic Profile 1.0. WebSphere Application Server conforms to the WS-I Basic Profile 1.1. UTF-16 is a kind of unicode encoding scheme using 16-bit values to store Universal Character Set (UCS) characters. UTF-8 is the most common encoding that is used on the Internet and UTF-16 encoding is typically used for Java and Windows product applications. You can change the encoding in a SOAP message from UTF-8 to UTF-16.

Before you begin

To learn more about the requirements of the Web Services-Interoperability Basic Profile (WS-I), including UTF-16, see Web Services-Interoperability Basic Profile information.

About this task

Support for UTF-16 encoding is required by WS-I Basic Profile. The application server only supports UTF-8 and UTF-16 encoding of SOAP messages.

You can change the character encoding in one of two ways:

Procedure

- Use a property on the Stub for users to set.

  This choice applies to the client only.

  For a client, the encoding is specified in the SOAP request. The SOAP engine serializes the request and sends it to the web service engine. The web service engine receives the request and deserializes the message to Java objects, which are returned to you in a response.

  When the web service engine on the server receives the serialized request, a raw message in the form of an input stream, is passed to the parser, which understands Byte Order Mark (BOM). BOM is mandatory for UTF-16 encoding and it can be used in UTF-8. The message is deserialized to a Java

objects and a service invocation is made. For two-way invocation, the engine needs to serialize the message using a specific encoding and send it back to the caller. The following example shows you how to use a property on the Stub to change the character set:

```
javax.xml.rpc.Stub stub=service.getPort("MyPortType");
((javax.xml.rpc.Stub)stub).setProperty(com.ibm.wsspi.webservices.Constants.MESSAGE_CHARACTER_SET_ENCODING,"UTF-16");
stub.invokeMethod();
```

In this code example, **com.ibm.wsspi.webservices.Constants.MESSAGE_CHARACTER_SET_ENCODING** = *"com.ibm.wsspi.webservices.xmlcharset";*

- Use a handler to change the character set through SOAP with Attachments API for Java (SAAJ).

  If you are using a handler, the SOAP message is transformed to a SAAJ format from other possible forms, such as an input stream. In such cases as a handleRequest method on the client side and a handleResponse method on the server side, the web services engine transforms from a SAAJ format back to the stream with appropriate character encoding. This transformation or change is called a *roundtrip transformation*. The following is an example of how you can use a handler to specify the character encoding through SAAJ:

```
handleResponse(MessageContext mc) {
 SOAPMessageContext smc = (SOAPMessageContext) context;
 javax.xml.soap.SOAPMessage msg = smc.getMessage();
 msg.setProperty (javax.xml.soap.SOAPMessage.CHARACTER_SET_ENCODING, "UTF-16");
 }
}
```

Results

You have modified the character encoding from UTF-8 to UTF-16 in the web service SOAP message.

Configuring the JAX-RPC web services client deployment descriptor with an assembly tool

You can configure JAX-RPC web services client deployment descriptor with an assembly tool.

Before you begin

You can configure deployment descriptors with assembly tools provided with WebSphere Application Server.

Also, you need an enterprise JavaBeans (EJB) Java archive (JAR) file, web application archive (WAR) file or an application client file that you can import into the assembly tool.

Assemble the client JAR file into an EAR file or assemble the client WAR file into an EAR file. To learn more, see the information on assembling a web services-enabled client JAR file into an EAR file or assembling a web services-enabled client WAR file into an EAR file.

About this task

Complete this task if you are developing a managed client that runs in the Java EE client container. This task is done after you assemble the EJB or web module.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate the web application archive (WAR) or Java Archive (JAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to assembly tools. To migrate

files, import your WAR or JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.

4. Configure the client deployment descriptor. Read about the configuring the client deployment descriptor in the Rational Application Developer information.

Results

You have a client deployment descriptor that is configured.

What to do next

Test the web services client. See the testing web services-enabled clients information to learn more about how to test an unmanaged client Java archive (JAR) file and an unmanaged client application.

Configuring the JAX-RPC client deployment descriptor for handler classes

You can configure the JAX-RPC client deployment descriptor for user-provided handler classes.

Before you begin

You need an enterprise archive (EAR) file for the applications that you want to configure. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file should contain the handler classes to configure. These classes implement the `javax.xml.rpc.handler.Handler` interface. For more information on writing handler classes, see chapter 6 of the Web Services for Java Platform, Enterprise Edition (Java EE) specification and chapter 12 of the Java API for XML-based remote procedure call (JAX-RPC) specification. The application modules must contain the `webservices.xml` (for server) and the client deployment descriptors.

For a complete list of the supported standards and specifications, see the web services specifications and API documentation.

About this task

Configure a handler in the client deployment descriptor by following the steps provided:

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer information.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer information.
3. Migrate the web application archive (WAR) or Java archive (JAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. Read about importing WAR or JAR files using an assembly tool in the Rational Application Developer information.
4. Configure the client deployment descriptor. Read about creating web services handlers in the Rational Application Developer information.

Results

You have a client deployment descriptor that is configured.

What to do next

Test the web services client. See the testing web services-enabled clients information to learn more about how to test an unmanaged client Java archive (JAR) file and an unmanaged client application.

Handler class properties with JAX-RPC

This article describes handler class properties using Java API for XML-based RPC (JAX-RPC).

You can configure the following handler class properties with assembly tools provided with WebSphere Application Server. See Configuring the `webservices.xml` deployment descriptor for Handler classes or Configuring the client deployment descriptors for Handler classes for instructions on how to configure the properties.

You can configure the following handler class properties with assembly tools provided with WebSphere Application Server. For instructions on how to configure the handler class properties, read about configuring the `webservices.xml` deployment descriptor for handler classes or configuring the client deployment descriptors for handler classes.

Description:

Standard Java Platform, Enterprise Edition (Java EE) technology descriptor field.

Display name:

Standard Java EE technology descriptor field.

Small icon:

Standard Java EE technology descriptor field.

Large icon:

Standard Java EE technology descriptor field.

Handler name:

The name of the handler. This name must be unique within the module.

Handler class:

The fully qualified name of the handler class. Initially, it is set by an assembly tool.

Initial parameters:

Property names and values available to the handler.

SOAP headers:

Qualified names (Qnames) of the SOAP headers that are processed by this handler.

See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) specification, available through the web services specifications and APIs information, for more information about setting this property.

SOAP roles:

URIs containing the SOAP actor names for which the handler acts in the role.

See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) specification, available through the web services specifications and APIs information, for more information about setting this property.

Example: Configuring handler classes for web services deployment descriptors

This scenario explains how to add a client and server handler class to a sample application, `WebServicesSamples.ear`. The handler classes display messages when given a request or response to handle.

The code for the client handler class is illustrated in the following example:

```
package samples;

public class ClientHandler implements javax.xml.rpc.handler.Handler {
    public ClientHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext context) {
    System.out.println("ClientHandler: In handleRequest");
    return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext context) {
    System.out.println("ClientHandler: In handleResponse");
    return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext context) {
    System.out.println("ClientHandler: In handleFault");
    return true;  }

public void init(javax.xml.rpc.handler.HandlerInfo config) {}

public void destroy() {}

public javax.xml.namespace.QName[] getHeaders() {
    return null; }
}
```

The code for the server handler class is illustrated in the following example:

```
package sample;
public class ServerHandler implements javax.xml.rpc.handler.Handler {
    public ServerHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext context) {
    System.out.println("ServerHandler: In handleRequest");
    return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext context) {
    System.out.println("ServerHandler: In handleResponse");
    return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext context) {
    System.out.println("ServerHandler: In handleFault");
    return true; }

public void init(javax.xml.rpc.handler.HandlerInfo config) {}

public void destroy() {}

public javax.xml.namespace.QName[] getHeaders() {
    return null;  }
}
```

1. Compile these classes using:
2. Open an assembly tool and import the two sample enterprise archive (EAR) files:
   -
   -
3. Import the compiled handler classes into the projects for the sample modules:
   - Import sample.ClientHandler into the **appClientModule** directory of the **AddressBookClient** project.
   - Import sample.ServerHandler into the **ejbModule** directory of the **AddressBookW2JE** project.
4. Configure the client deployment descriptor for handler classes.

This topic explains how to configure the client deployment descriptor for user-provided handler classes.

5. Configure the webservices.xml deployment descriptor for handler classes.

   This topic explains how to configure the `webservices.xml` deployment descriptor for user-provided handler classes.

6. Save your changes and export the EAR files.
7. Uninstall the `WebServicesSamples.ear` application from your server if it is already installed.
8. Install the new `WebServicesSamples.ear` application.
9. Start the server.
10. Run the client:

    ```
    launchClient ApplicationClients.ear -CCjar=AddressBookClient.jar
    ```

    When the client runs, the console output looks like the following example. The messages from the handlers are shown in bold.

```
IBM WebSphere Application Server
J2EE Application Client Tool
Copyright IBM Corp., 1997-2003
WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client
Environment.
WSCL0035I: Initialization of the J2EE Application Client
Environment has completed.
WSCL0014I: Invoking the Application Client class
com.ibm.websphere.samples.webservices.addr.AddressBookClient
>> Querying address for 'Purdue Boilermaker' using port
AddressBookW2JE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       1 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       2 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       3 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port AddressBookW2JB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       4 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
```

    For the client, the handler class is configured for each service reference, not for each port. The AddressBook sample has four ports, but only one service reference, therefore the ClientHandler handles requests and responses on all ports.

When the server log file is examined, it contains the following data:

```
[9/24/03 16:39:22:661 CDT] 4deec1c6 WebGroup      I SRVE0180I:
[HTTP router for AddressBookW2JE.jar] [/AddressBookW2JE] [Servlet.LOG]:
AddressBook: init
[9/24/03 16:39:23:161 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleRequest
[9/24/03 16:39:23:211 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleResponse
```

Results

The deployment descriptors for handler classes are configured. Deployment descriptors are required so that so that WebSphere Application Server can process the incoming web services requests.

What to do next

Deploy the EAR file that has been configured and enabled for web services. Then you can test the application to make sure it runs within the WebSphere Application Server environment.

Configuring the JAX-RPC web services client bindings in the ibm-webservicesclient-bnd.xmi deployment descriptor

You can configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file with assembly tools.

Before you begin

You can configure deployment descriptors with assembly tools provided with the application server.

You must configure the assembly tool before you can use it. Read about configuring the assembly tool in the Rational Application Developer documentation.

About this task

Now that you have assembled the client module, complete this step to configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor. Deployment descriptors are required so that so that WebSphere Application Server can process the incoming web services requests.

Configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file with the following steps provided:

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. Switch to the Java EE Perspective.
   a. Click **Window** > **Open Perspective** > **Other** > **Java EE**.
3. Open the Project Explorer.
   a. Click **Window** > **Show View** > **Other** > **Project Explorer**.
4. Locate the deployment descriptor file for the module. Hint: Deployment Descriptor: *<module>*
5. Double-click the deployment descriptor file to open the Deployment Descriptor editor.
   a. Select the **WS Binding** tab at the bottom of the editor window to open the Web Services Client Bindings editor.
6. Verify the serviceRefLink element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Click the **Services References** tab.
   c. Click **Add**.
   d. Select the service references defined in the client deployment descriptor file from the list.
7. Verify the deployedWSDLFile element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Select the service reference.
   c. Expand the **Service Reference Details** section.
   d. Click **Browse** that is located to the right of the Deployed WSDL file field.

e. Select the new Web Services Description Language (WSDL) file.
f. Click **OK**.

You can also change the deployedWSDLFile element of a deployed web service using the administrative console. Click **Enterprise Applications** > ***application*** > **Web module or EJB module** > ***module*** > **Web services client bindings**.

8. Verify the `defaultMappings` element settings.
   a. Open the **Web services client bindings** editor.
   b. Click **Default mappings**.
   c. Click **Add**.
   d. Edit the entries in the newly added row to establish a mapping between a *portType* and a *port* in the WSDL file. Only one entry is supported for each *portType*.
   e. Click **OK**.

   You can also change the defaultMappings element of a deployed web service using the administrative console. Click **Enterprise Applications** > ***application*** > **Web module or EJB module** > ***module*** > **Web services client bindings**.
9. Access the web services client Port bindings editor through the **Port qualified name binding details** section at the bottom of the editor pane.
10. Verify the syncTimeout element settings.
    a. Create a Port qualified name bindings for the port.
    b. Open the **Web services client bindings** editor.
    c. Confirm that a service reference is selected in either the **Component-scoped references** or the **Service references** section.
    d. Expand the **Port qualified name binding** section.
    e. Click **Add**. The **Add port qualified name binding** dialog opens.
    f. Type the *namespace* of the WSDL file port you want to configure, in the **Port namespace link** field.
    g. Type the *local_name* of the WSDL file port you want to configure in the **Port local name link** field. The name displayed in the **Port qualified name binding** list is the local name of the WSDL file port.
    h. Click **OK**.
    a. Configure the syncTimeout property by locating the Synchronization timeout field and enter the desired value. The default is 300 seconds.
11. Verify the basicAuth element settings.
    a. Locate the **HTTP basic authentication** field in the **Port qualified name binding details** section.
    b. Type the desired value in the User ID and Password fields.
    c. Click **OK**.
12. Verify the sslConfig element settings.
    a. Locate the **SSL configuration** field in the **Port qualified name binding details** section.
    b. Type the desired value in the **Name** field.
    c. Click **OK**.
13. After editing the properties, type `ctrl-s` on your keyboard to save the changes.

Results

You have configured the `ibm-webservicesclient-bnd.xmi` deployment descriptor. If you have configured all of the client deployment descriptors, test the web services client. If you have not configured all of the client deployment descriptors, complete the configurations and then test the web services client.

ibm-webservicesclient-bnd.xmi assembly properties for JAX applications

The ibm-webservicesclient-bnd.xmi deployment descriptor file contains information for the web services run time that is WebSphere product-specific. This deployment descriptor file is used with Java API for XML-based web services.

You can configure deployment descriptors with assembly tools provided with WebSphere Application Server. Read about configuring the JAX-RPC web services client bindings in the ibm-webservicesclient-bnd.xmi deployment descriptor to learn more about configuring this deployment descriptor.

Assembly properties

The following list is a collection of supported properties and attributes that you can define for your applications.

Note: The overriddenEndpointURI property is the only property that is applicable for Java API for XML-Based Web Services (JAX-WS) Web services.

componentNameLink
An attribute of the componentScopedRefs element. When a web service is implemented by an Enterprise JavaBeans (EJB) implementation, each <componentScopedRefs> element contains assembly properties for an individual enterprise bean. The componentNameLink attribute of the <componentScopedRefs> element identifies the enterprise bean that the assembly properties apply to by specifying the <ejb-name>. This property is used only when the web service client is an enterprise bean.

serviceRefLink
An attribute of the serviceRefs element. Specifies the link to the <service-ref-name> in the <service-ref> element in the client deployment descriptor. The client deployment descriptor is either `ejb-jar.xml`, `web.xml` or `application-client.xml`.

deployedWSDLFile
An attribute of the serviceRefs element is optional. Permits an alternate Web Services Description Language (WSDL) file to use other than that specified in the <wsdl-file> element of the <service-ref> element in the client deployment descriptor. If an attribute is specified, the alternate WSDL file must be packaged in the same module and must be compatible with the development WSDL file. The deployedWSDLFile property supplies a new WSDL file containing a different endpoint web address than the original WSDL file.

defaultMappings
An element that identifies which port to use for a given portType when one is not selected by the client. This element has the following attributes: portTypeNamespace, portTypeLocalName, portNamespace, portLocalName. These attributes identify which wsdl:port is used for a wsdl:portType.

syncTimeout
An attribute of the portQnameBindings element. Specifies how long, in seconds, to wait for a response from a synchronous call. The default is 300 seconds.

basicAuth
An element of the portQnameBindings element. Authenticates a service client to the service endpoint, independent of the underlying transport that includes, HTTP, HTTPS, and Java Message Service (JMS). Set the user ID and password attributes as needed.

sslConfig
An element of the portQnameBindings element. Specifies the Secure Sockets Layer (SSL) configuration of an HTTPS outbound request. The name attribute is the name of an SSL configuration entry or alias that is defined in the SSL configuration repertoire. This attribute is used only when the client is running in the WebSphere Application Server.

For WebSphere Application Server for z/OS, some digital certificate and keyring management is required. To learn more, see the creating Secure Sockets Layer digital certificates and System Authorization Facility keyrings that applications can use to initiate HTTPS requests information.

overriddenEndpointURI
A property that specifies the final URL to which requests will be sent. When the client is run to invoke the web service, the request is sent to this property's URL, which takes precedence over the address given in the WSDL file and the client code. This property is the only property in the ibm-webservicesclient-bnd.xmi file which is applicable to JAX-WS services.

A bindings file example

The following example demonstrates the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute whose value is a unique string. Review the template `xmi` files generated by the **WSDL2Java** command for examples of ID strings. Read about the **WSDL2Java** command-line tool for Java API for XML-based Remote Procedure Call (JAX-RPC) applications to learn more about this tool.

```
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:com.ibm.etools.webservice.wscbnd=
"http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">

 <componentScopedRefs componentNameLink="myComponent ref"/>

 <serviceRefs serviceRefLink="myService ref" deployedWSDLFile="META-INF/wsdl/alternate.wsdl">
  <defaultMappings portTypeLocalName="AddressBook" portTypeNamespace="http://www.com.ibm"
portLocalName="AddressBookPort" portNamespace="http://www.com.ibm"/>
   <portQnameBindings portQnameNamespaceLink="http://www.com.ibm"
portQnameLocalNameLink="AddressBookPort" syncTimeout="99">
    <basicAuth userid="myId" password="myPassword"/>
    <sslConfig name="mynode/DefaultSSLSettings"/>
   </portQnameBindings>
  </serviceRefs>
 </com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Implementing extensions to JAX-RPC web services clients

WebSphere Application Server provides extensions to web services clients using the Java API for XML-based RPC (JAX-RPC) programming model.

About this task

You can customize web services by using the following extensions to the JAX-RPC client programming model.

Procedure

- Set the **REQUEST_SOAP_HEADERS** and **RESPONSE_SOAP_HEADERS** properties in a JAX-RPC client Stub to enable a web services client to send or retrieve implicit SOAP headers.

  An implicit SOAP header is a SOAP header that is not explicitly defined in the WSDL file. An implicit SOAP header file fits one of the following descriptions:

  – A message part that is declared as a SOAP header in the binding in the WSDL file, but the message definition is not referenced by a portType within a WSDL file.
  – An element that is not contained in the WSDL file.

  Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

  To learn how to modify your client code to send or retrieve transport headers, see the information on sending implicit SOAP headers with JAX-RPC or receiving implicit SOAP headers with JAX-RPC.
- Set the **REQUEST_TRANSPORT_PROPERTIES** and **RESPONSE_TRANSPORT_PROPERTIES** properties to enable a web services client to send or retrieve transport headers.

Set the properties on the Stub or Call object.

By modifying your client code to send or retrieve transport headers, you can send or receive specific information within the transport headers of outgoing requests or incoming responses from the server. For requests or responses that use the HTTP transport, the information is sent or retrieved in an HTTP header. Similarly, for a request or response that uses the Java Message Service (JMS) transport, the information is sent or retrieved in a JMS message property.

To learn how to modify your client code to send or retrieve transport headers, see the information on sending transport headers with JAX-RPC or retrieving transport headers with JAX-RPC.

To learn how to enable a Web services client to send or retrieve transport headers, see the transport header properties best practices information.

- Implement support for `javax.xml.rpc.ServiceFactory.loadService()` methods.

  The loadService methods create an instance of the generated service implementation class in an implementation-specific manner. The loadService methods are new for JAX-RPC 1.1 and include three signatures:

  – **public.javax.xml.rpc.Service loadService (Class serviceInterface)**

    As documented in the JAX-RPC specification, this method returns the generated service implementation for the service interface. See the web services specifications and API documentation to review the JAX-RPC specification.

  – **public.javax.xml.rpc.Service loadService (URL wsdlDocumentLocation, Class serviceInterface, Properties properties)**

    This method behaves like the loadService (Class serviceInterface) because the following parameters are ignored:

    - wsdlDocumentLocation
    - properties

  – **public.javax.xml.rpc.Service loadService (URL wsdlDocumentLocation, QName serviceName, Properties properties)**

    This method returns the generated service implementation for the specified service by using optional namespace-to-package mapping information.

    - wsdlDocumentLocation - ignored
    - serviceName - QName (namespace, localpart) of the service
    - properties - If this parameter is non-null, it contains namespace-to-package mapping entries. Each Property entry key is a String corresponding to the namespace. Each Property entry value is a String corresponding to the Java package name.

      If the properties argument contains an entry with a key (namespace) that matches the namespace portion of the QName serviceName argument, the entry value (`javaPackage`) is used as the package name when trying to locate the service implementation.

  For more information on these methods, see the JAX-RPC specification.

- Implement the `CustomBinder` interface to provide concrete custom data binders for a specific XML schema type **(JAX-RPC applications only)**.

  Custom data binders are used to map XML schema types with Java objects. Custom data binders provide bindings for XML schema types that are not supported by the current Java API for XML-based Remote Call Procedure (JAX-RPC) specification. WebSphere Application Server provides an extension to the Web Services for Java Platform, Enterprise Edition (Java EE) programming model called the CustomBinder interface that implements these custom bindings for a specific XML schema type. The CustomBinder interface has three properties, in addition to deserialize and serialize methods:

  – QName for the XML schema type

  – QName scope

  – Java type

  The custom data binder defines serialize and deserialize methods to convert between a Java object and a SOAPElement interface. A custom data binder is added to the runtime system and interacts with the

web services runtime using a SOAPElement. They are added to the runtime by using custom binding providers. Read about the custom data binders and the custom binding provider to learn more. See the CustomBinder interface documentation to learn more about how you can implement this interface to provide concrete custom data binders for a specific XML schema type.

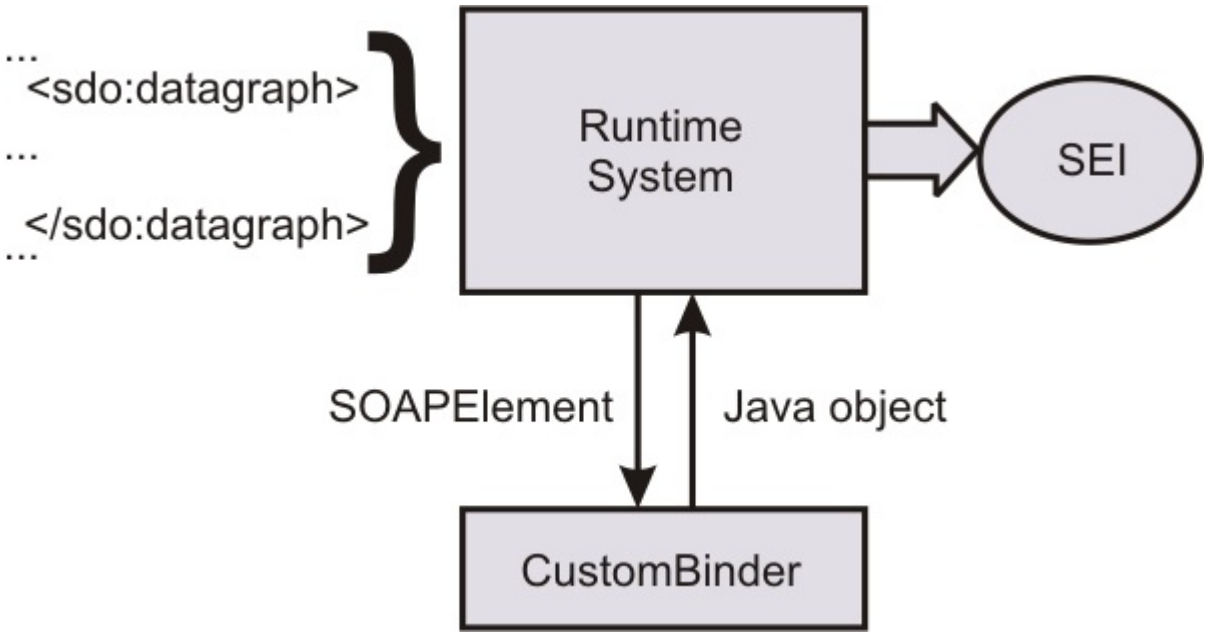
Custom data binders for JAX-RPC applications

A *custom data binder* is used to map XML schema types with Java objects. Custom data binders provide bindings for XML schema types that are not supported by the current Java API for XML-based Remote Call Procedure (JAX-RPC) specification.

The custom data binder defines serialize and deserialize methods to convert between a Java object and a SOAPElement interface. A custom data binder is added to the runtime system and interacts with the web services run time using a SOAPElement. Unlike conventional deserializers, custom data binders do not rely on the low-level parsing events from the run time to build the Java object, such as Simple API for XML (SAX). Instead, the run time builds the custom data binder by rendering the incoming SOAP message into a SOAPElement. The SOAPElement that contains the message is passed to the customer data binder. For example, if the incoming message contains a Service Data Object (SDO) datagraph, the runtime system processes as follows:

1. The runtime system recognizes the `<sdo:Datagraph>` code.
2. The run time queries the type mapping system to locate the custom data binder for the datagraph data, for example SDOCustomBinder.
3. A SOAPElement is created that represents the incoming SDO datagraph.
4. The run time passes the SOAPElement to the SDOCustomBinder.

Within the deserialized method, the SDOCustomBinder extracts the content from the SOAPElement and builds a concrete DataGraph object with a `commonj.sdo.DataGraph` type.



When a Java object is serialized, a similar process occurs. The run time locates a custom data binder and converts the Java object to a SOAPElement. The runtime serializes the SOAPElement to the raw message that is transported in the output stream.

The following is an example of an XML schema that is defined by the SDO specification:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sdo="commonj.sdo"
  targetNamespace="commonj.sdo">

  <xsd:element name="datagraph" type="sdo:DataGraphType"/>

  <xsd:complexType name="DataGraphType">
  <xsd:complexContent>
    <xsd:extension base="sdo:BaseDataGraphType">
      <xsd:sequence>
        <xsd:any minOccurs="0" maxOccurs="1"
                 namespace="##other" processContents="lax"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="BaseDataGraphType" abstract="true">
  <xsd:sequence>
    <xsd:element name="models" type="sdo:ModelsType" minOccurs="0"/>
    <xsd:element name="xsd" type="sdo:XSDType" minOccurs="0"/>
    <xsd:element name="changeSummary"
                 type="sdo:ChangeSummaryType" minOccurs="0"/>
   </xsd:sequence>
  <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

  <xsd:complexType name="ModelsType">
  <xsd:sequence>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
             namespace="##other" processContents="lax"/>
  </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="XSDType">
  <xsd:sequence>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
        namespace="http://www.w3.org/2001/XMLSchema" processContents="lax"/>
  </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ChangeSummaryType">
  <xsd:sequence>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
             namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:attribute name="create" type="xsd:string"/>
  <xsd:attribute name="delete" type="xsd:string"/>
  </xsd:complexType>

</xsd:schema>
```

WebSphere Application Server defines the CustomBinder interface that implements concrete custom bindings for a specific XML schema type.

The custom binding provider is used to import the custom bindings into the run time. To learn how to plug your custom data binders into the **WSDL2Java** command-line tool for development, read about custom binding providers. You can also read about usage patterns for deploying custom data binders to learn more about how to deploy the provider package to your runtime, as well as the roles involved in the custom binding process.

Custom binding providers for JAX-RPC applications

A *custom binding provider* is the packaging of custom data binder classes with a declarative metadata file. The main purpose of a custom binding provider is to aggregate related custom data binders to support particular user scenarios. The custom binding provider is used to plug the custom data binders into the emitter tools and the run time system so that the emitter tools can generate the appropriate artifacts and the run time system can augment its existing type mapping system to reflect the applied custom data binders and invoke them.

A custom binding provider works with a specific XML schema type, while applications involve a few related XML schema types. You need a mechanism to aggregate and declare various custom data binders to provide a complete binding solution. The concept of the custom binding provider defines a declarative model that can be used to plug in a set of custom data binders to either emitter tools or the run time system.

See the custom data binders information and the information on the CustomBinder interface to learn more about custom data binders and the CustomBinder API included in WebSphere Application Server to define the custom data binders. After you have defined the custom data binders, you are ready to deploy the custom binder package. To learn how to deploy this package, see the information on usage patterns for deploying custom data binders for JAX-RPC applications.

The declarative metadata file, `CustomBindingProvider.xml`, is an XML file that is packaged with the custom provider classes in a single Java archive (JAR) file and located in the `/META-INF/services/directory`. Once a provider JAR file is packaged, the binary information and the metadata file located in the JAR file can be used by the **WSDL2Java** command-line tool and the run time system.

The following example is the XML schema for the `CustomBindingProvider.xml` file. The top-level type is the providerType that contains a list of mapping elements. Each mapping element defines the associated custom data binder and properties, including xmlQName, javaName and qnameScope. You can read more about these properties in the information for CustomBinder interface for JAX-RPC applications. The providerType also has an attribute called scope that has a value of *server*, *application* or *module*. The scope attribute is used by the server deployment to resolve the conflict and to realize a custom binding hierarchy.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    targetNamespace=
       "http://www.ibm.com/webservices/customdatabinding/2004/06"
    xmlns:customdatabinding=
        "http://www.ibm.com/webservices/customdatabinding/2004/06"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="qualified">

   <xsd:element name="provider" type="customdatabinding:providerType"/>

   <xsd:complexType name="providerType">
   <xsd:sequence>
     <xsd:element name="description" type="xsd:string" minOccurs="0"/>
     <xsd:element name="mapping" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="description" type="xsd:string" minOccurs="0"/>
          <xsd:element name="xmlQName" type="xsd:QName"/>
          <xsd:element name="javaName" type="xsd:string"/>
          <xsd:element name="qnameScope"
                       type="customdatabinding:qnameScopeType"/>
          <xsd:element name="binder" type="xsd:string"/>
        </xsd:sequence>
       /xsd:complexType>
      </xsd:element>
       <xsd:attribute name="scope"
            type="customdatabinding:ProviderScopeType" default="module"/>
      </xsd:sequence>
   </xsd:complexType

  <xsd:simpleType name="qnameScopeType">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="simpleType"/>
         <xsd:enumeration value="complexType"/>
         <xsd:enumeration value="element"/>
      </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ProviderScopeType">
     <xsd:restriction base="xsd:string">
         <xsd:enumeration value="server"/>
         <xsd:enumeration value="application"/>
```

```
        <xsd:enumeration value="module"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

The following is an example of the `CustomBindingProvider.xml` file for the SDO DataGraph schema that was introduced in CustomBinder interface. The example displays the mapping between a schema type, DataGraphType, and a Java type, commonj.sdo.DataGraph. The binder that represents this mapping is called `test.sdo.SDODataGraphBinder`.

```
<cdb:provider
 xmlns:cdb="http://www.ibm.com/webservices/customdatabinding/2004/06"
   xmlns:sdo="commonj.sdo">
   <cdb:mapping>
     <cdb:xmlQName>sdo:DataGraphType</cdb:xmlQName>
     <cdb:javaName>commonj.sdo.DataGraph</cdb:javaName>
     <cdb:qnameScope>complexType</cdb:qnameScope>
     <cdb:binder>test.sdo.SDODataGraphBinder</cdb:binder>
   </cdb:mapping>
</cdb:provider>
```

You need to import your custom data binders into the **WSDL2Java** command-line tool for development purposes. The custom data binders affect how the development artifacts, including the Service Endpoint Interface and the JSR 109 mapping data, are generated from the Web Services Description Language (WSDL) file. The **WSDL2Java** command-line tool ships with WebSphere Application Server and uses the custom binder Java archive file, or custom binder package, to generate these the development artifacts.

The following example is a WSDL file that references the SDO DataGraph schema that is introduced in the CustomBinder interface topic.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://sdo.test"
   xmlns:impl="http://sdo.test"
   xmlns:intf="http://sdo.test"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sdo="commonj.sdo">

 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://sdo.test"
       xmlns="http://www.w3.org/2001/XMLSchema" xmlns:sdo="commonj.sdo">
   <import namespace="commonj.sdo" schemaLocation="sdo.xsd"/>
  </schema>
 </wsdl:types>

 <wsdl:message name="echoResponse">
  <wsdl:part element="sdo:datagraph" name="return"/>
 </wsdl:message>

 <wsdl:message name="echoRequest">
   <wsdl:part element="sdo:datagraph" name="parameter"/>
 </wsdl:message>

 <wsdl:portType name="EchoService">
   <wsdl:operation name="echo">
     <wsdl:input message="impl:echoRequest" name="echoRequest"/>
     <wsdl:output message="impl:echoResponse" name="echoResponse"/>
    </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="EchoServiceSoapBinding" type="impl:EchoService">
   <wsdlsoap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http"/>
     <wsdl:operation name="echo">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="echoRequest">
         <wsdlsoap:body use="literal"/>
        </wsdl:input>

       <wsdl:output name="echoResponse">
          <wsdlsoap:body use="literal"/>
       </wsdl:output>
    </wsdl:operation>
```

```
  </wsdl:binding>

  <wsdl:service name="EchoServiceService">
     <wsdl:port binding="impl:EchoServiceSoapBinding" name="EchoService">
       <wsdlsoap:address location="http://<uri>"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

If you run the **WSDL2Java** command without the custom data binding package, the following Service Endpoint Interface is generated with a parameter type, as dictated by the JAX-RPC specification:

```
public interface EchoService extends java.rmi.Remote {
  public javax.xml.soap.SOAPElement
   echo(javax.xml.soap.SOAPElement parameter)
    throws java.rmi.RemoteException;
}
```

When you run the **WSDL2Java** command with the custom data binding package, the custom data binders are used to generate the parameter types. To apply the custom data binders, use the `-classpath` option on the WSDL2Java tool. The tool searches its classpath to locate all the files with the same file path of `/META-INF/services/CustomBindingProvider.xml`. The following is an example how you can use the command to generate a Service Endpoint Interface with the parameter type of `commonj.sdo.Datagraph`:

```
WSDL2Java -role develop-server -container web classpath sdobinder.jar echo.wsdl
```

The Service Endpoint Interface that is generated looks like the following:

```
public interface EchoService extends java.rmi.Remote {
  public commonj.sdo.DataGraph
   echo(commonj.sdo.DataGraph parameter)
    throws java.rmi.RemoteException;
}
```

The custom binder packaged JAR file has to be made available at runtime to make sure the web service client is invoked, regardless if it is a stub-based client or a Dynamic Invocation Interface (DII) client. The same applies to the service.

CustomBinder interface for JAX-RPC applications

WebSphere Application Server defines a CustomBinder interface that you can implement for Java API for XML-based Remote Call Procedure (JAX-RPC) applications to provide concrete custom data binders for a specific XML schema type.

The CustomBinder interface has three properties, in addition to deserialize and serialize methods. These properties are QName for the XML schema type, the QName scope, and the Java type that the schema type maps to. The properties are accessible through the corresponding getter methods.

getQName

The getQName method returns the QName of the target XML schema type. Custom data binders only work with the root level schema type.

For anonymous types, the getQName method returns the QName of the containing element.

For named types, the getQName method returns the QName of the complexType or the simpleType.

getQNameScope

The getQNameScope method returns the binder qnameScope property that indicates whether the schema type is a named type or an anonymous type. The qnameScope property value can be *complexType* for an <xsd:complexType>, *simpleType* for an <xsd:simpleType> or *element* for an <xsd:element> that is defined with an anonymous type.

In the following schema, data1 is an element that is defined with an anonymous type. The element, data2, is defined using the named type, data2Type.

```
<xsd:element name="data1">
 <xsd:complexType>
  ...
 </xsd:complexType>
</xsd:element>

<xsd:element name= "data2" type="data2Type"/>
<xsd:complexType name="data2Type">
 ...
</xsd:complexType>
```

The anonymous type, data1, has a qNameScope of element and a qName of data1. The type, data2Type, has a qNameScope of complexType and a qName of data2Type.

The element, data2, is not represented in the custom data binder. The custom data binder only processes types and not elements.

getJavaName

The getJavaName method returns the fully-qualified class name for the Java type that is mapped to the named or anonymous type. The class can be an interface or a concrete class. The object returned from the deserialize method has a type that is compatible with the Java type that is returned by the getJavaName method.

serialize

The serialize method returns the SOAPElement that the custom data binder builds from the Java object. The Java object is passed from the run time system and is expected to match what is returned from the getJavaName method. The SOAPElement parameter does not have child elements, but it does have a valid QName. This parameter is a reference for the binder to create the final SOAPElement.

In most cases, the binder implementation appends the child elements to the root SOAPElement. The run time system guarantees that the SOAPElement QName is correct. Therefore, the custom data binder for named types keeps the QName of the root element because the binder does not know the enclosing element. The binder implementation for an anonymous type should always include the QName in the returned SOAPElement that matches the defined schema type. WebSphere Application Server does not have concrete methods in the CustomBindingContext parameter.

deserialize

The deserialize method returns a Java object that the custom data binder builds from the passed root SOAPElement. The object type of the returned Java object must match what is returned from the getJavaName method. Unlike the parameter serialize method, the passed SOAPElement contains the original XML data with the necessary namespace declarations.

The following is an example of an implementation of the SDO DataGraph binder, where the convertToSDO and convertToSAAJ utility methods convert between SOAPElement and an SDO object.

```
package test.sdo.binder;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;

import com.ibm.wsspi.webservices.binding.CustomBinder;
import com.ibm.wsspi.webservices.binding.CustomBindingContext;

public class DataGraphBinder implements CustomBinder {
 public QName getQName() {
  return new QName("commonj.sdo", "DataGraphyType");
}
public String getJavaName() {
```

```
 return CustomBinder.QNAME_SCOPE_COMPLEXTYPE;
}
public String getJavaName() {
 return commonj.sdo.DataGraph.class.getName();
}
public javax.xml.soap.SOAPElement serialize(
  Object bean,
  SOAPElement rootNode,
  CustomBindingContext context)
  throws javax,xml.soap.SOAPException {
 // convertToSAAJ is a utility method to convert
 // the SDO DataGraph to the SOAPElement
 return convertToSAAJ(bean, rootNode);

public Object deserialize(
  SOAPElement source,
  CustomBindingContext context)
  throws javax.xml.soap.SOAPException {
  // convertToSDO is a utility method to convert
  // the SOAPElement to the SDO DataGraph
  return convertToSDO(source);
 }
}
```

To learn more about custom data binders, see the custom data binders for JAX-RPC applications information. To learn how to plug your custom data binders into the **WSDL2Java** command-line tool for development, see the custom binding providers for JAX-RPC applications information.

Usage patterns for deploying custom data binders for JAX-RPC applications

Custom data binders are used to map XML schema types with Java objects. Custom data binders provide bindings for XML schema types that are not supported by the current Java API for XML-based Remote Call Procedure (JAX-RPC) specification. WebSphere Application Server provides an extension to the Web Services for Java Platform, Enterprise Edition (Java EE) programming model called the CustomBinder interface that implements these custom bindings for a specific XML schema type. The custom binding provider is the package for the custom data binders that is imported into the runtime.

To learn more about the CustomBinder API, see the CustomBinder interface for JAX-RPC applications information. For general information about custom binders, see the custom data binders for JAX-RPC applications information. See the custom binding providers for JAX-RPC applications information to review how custom binding providers are packaged for development.

This usage pattern reviews how to deploy the provider package to your runtime, as well as the roles involved in the custom binding process.

Roles involved in custom data binding

Four roles are involved with custom data binding. These roles that are defined by the Web Services for Java Platform, Enterprise Edition (Java EE) specification are as follows:

- **Custom binding provider** is responsible for implementing the required custom data binders, declaring these binders in a `CustomBindingProvider.xml` file and packaging the binding classes into a Java archive (JAR) file.
- **Application developer** is responsible for applying the custom binding provider JAR file and generating the development artifacts.
- **Application assembler** needs to understand the application requirements in terms of the custom data binding and decides how to package the custom provider JAR file as a part of the application.
- **Application deployer** configures the shared libraries to make custom data binding support available to the applications. This needs to be done if the custom provider JAR file is not packaged with the application. If the application is not deployed, the deployer has to run the web services deployment tools after the application is installed.

Common usage patterns

The custom binder provider package can be deployed in various ways to provide flexibility beyond the standard JAX-RPC mapping standards. Three primary deployment usage patterns are as follows:

- **Deploy the custom data binders at the server level**

  This pattern ensures that all the applications that are running on the server are affected by the custom data binders and is useful if fundamental XML types are introduced but are not supported by the standard JAX-RPC mapping rules.

  This type of situation occurs frequently for new web services specifications that define new schema types. For example, the WS-Addressing specification defines an EndpointReferenceType schema type that is not supported by the JAX-RPC mapping rules. Because this pattern requires augmenting the server classpath, it has a significant impact on the server runtime and affects the installed applications. This pattern is most suitable for WebSphere Application Server internal components.
- **Deploy the custom binders for one or more application**

  Use this pattern if you only want specified applications to be affected by the custom data binders and if relevant XML schema types apply to a set of applications. You can share the custom data binders within a set of applications while achieving isolation between different sets of applications.
- **Deploy the custom binders for a specific web module within an application**

  Using this pattern ensures that a specific Web module is affected by the deployed custom data binders. This pattern is useful when fine granularity for custom binding is required. You cannot use this pattern with EJB modules because the module and its referenced library belong to the entire application.

Usage patterns

This section reviews deploying custom data binders using one of the three patterns:

- **Server level deployment**

  If you deploy the custom data binders at the server level, you need to set the scope attribute of the declared binding provider as *server*. Setting the value to *server* guarantees a higher priority for declared binders if there are conflicts between the server and applications. The custom binding provider JAR file needs to be in the appropriate place to be picked up by the server runtime. Configure the server path and make the custom binding provider JAR file a part of the server classpath. To learn about values used in configuring the server classpath, see the Java virtual machine settings information.
- **Deploying custom data binders for one or more applications**

  To deploy custom data binders for one or more applications, set the scope attribute of the declared custom binding provider as *application*. Setting the value to application guarantees higher priority binders in case of conflicts between the application and the module. If the custom data binders are used by more than one application, configure a shared library for the applications to reference. To learn about values used in configuring the shared libraries path, see the managing shared libraries information.
- **Deploy the custom data binders for a specific web module within an application**

  To deploy custom data binders for a specific web module within an application, set the scope attribute of the declared custom binding provider to the value *module*. The only way to apply the custom data binder for this pattern is to pre-package the custom binding provider JAR file with the web module, for example, place the JAR file in the `/WEB-INF/lib` directory.

Sending implicit SOAP headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to send values in implicit SOAP headers. By modifying your client code to send implicit SOAP headers, you can send specific information within an outgoing web service request.

Before you begin

To complete this task, you need a web services client that you can enable to send implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

You cannot manipulate protected SOAP headers. A SOAP header that is declared protected by its owning component, for example, Web Services Security, is not accessible to client applications. An exception occurs if you try to manipulate protected SOAP headers.

About this task

The client application sets properties on the Stub or Call object to send and receive implicit SOAP headers.

Procedure

1. Create a java.util.HashMap object.
2. Add an entry to the HashMap object for each implicit SOAP header that the client wants to send. The HashMap entry key is the QName of the SOAP header. The HashMap entry value is either an SAAJ SOAPElement object or a String that contains the XML text of the entire SOAP header element.
3. Set the HashMap object as a property on the Stub or Call object. The property name is com.ibm.websphere.webservices.Constants.REQUEST_SOAP_HEADERS. The value of the property is the HashMap.
4. Issue the remote method calls using the Stub or Call object. The headers within the HashMap object are sent in the outgoing message.

   A `JAXRPCException` error can occur if any of the following are true:

   - The HashMap object contains a key that is not a QName object or if the HashMap object contains a value that is not a String or a SOAPElement object.
   - The HashMap object contains a key that represents a SOAP header that is declared protected by the owning component.

Results

You have a JAX-RPC web services client that is configured to send implicit SOAP headers.

Example

The following programming example illustrates how to send two request SOAP headers and receive one response SOAP header within a web services request and response:

```
1 //Create the request and response hashmaps.
2 HashMap requestHeaders=new HashMap();
3 HashMap responseHeaders=new HashMap();
4
5 //Add "AtmUuid1" and "AtmUuid2" to the request hashmap.
6 requestHeaders.put(new QName("com.rotbank.security", "AtmUuid1"),
7   "<AtmUuid1 xmlns=\"><uuid>ROTB-0A01254385FCA09</uuid></AtmUuid1>");
8 requestHeaders.put(new QName("com.rotbank.security", "AtmUuid2"),
9  ((IBMSOAPFactory)SOAPFactory.newInstance()).createElementFromXMLString(
10   "x:AtmUuid2 xmlns:x=\"com.rotbank.security\"><x:uuid>ROTB-0A01254385FCA09
```

```
    </x:uuid><x:AtmUuid2>"));
11
12 //Add "ServerUuid" to the response hashmap.
13 //If "responseHeaders" is empty, all the SOAP headers are
14 //extracted from the response message.
15 responseHeaders.put(new QName("com.rotbank.security","ServerUuid"), null);
16
17 //Set the properties on the Stub object.
18 stub.setProperty(Constants.REQUEST_SOAP_HEADERS.requestHeaders);
19 stub.setProperty(Constants.RESPONSE_SOAP_HEADERS.responseHeaders);
20
21 //Call the operationon the Stub.
22 stub.foo(parm2, parm2);
23
24 //Retrieve "ServerUuid" from the response hashmap.
25 SOAPElement serverUuid =
26   (SOAPElement) responseHeaders.get(new QName("com.rotbank.security","ServerUuid"));
27
28 //Note: "serverUuid" now equals a SOAPElement object that represents the
29 //following code:
30//"<y:ServerUuid xmlns:y=\"com.rotbank.security\"><:uuid>ROTB-0A03519322FSA01
   </y:uuid></y:ServerUuid.");
```

On lines 2-3, new HashMaps are created that are used for the request and response SOAP headers.

On lines 6-10, the AtmUuid1 and AtmUuid2 headers elements are added to the request HashMap.

On line 15, the ServerUuid header element name, along with a null value, is added to the response HashMap.

On line 18, the request HashMap is set as a property on the Stub object. This causes the AtmUuid1 and AtmUuid2 headers to be added to each request message that is associated with an operation that is invoked on the Stub object.

On line 19, the response HashMap is set as a property on the Stub object. This causes the ServerUuid header to be extracted from each response message that is associated with an operation that is invoked on the Stub object.

On line 22, the web service operation is invoked on the Stub object.

On lines 25-26, the ServerUuid header is retrieved from the response HashMap. The header was extracted from the response message and inserted into the HashMap by the web services engine.

Receiving implicit SOAP headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to receive values from implicit SOAP headers. By modifying your client code to receive implicit SOAP headers, you can receive specific information within an incoming web service response.

Before you begin

To complete this task, you need a web services client that you can enable to receive implicit SOAP headers.

An *implicit SOAP header* is a SOAP header that fits one of the following descriptions:

- A message part that is declared as a SOAP header in the binding in the Web Services Description Language (WSDL) file, but the message definition is not referenced by a portType element within a WSDL file.
- An element that is not contained in the WSDL file.

Handlers and service endpoints can manipulate implicit or explicit SOAP headers using the SOAP with Attachments API for Java (SAAJ) data model.

You cannot manipulate protected SOAP headers. A SOAP header that is declared protected by its owning component, for example, Web Services Security, is not accessible to client applications. An exception occurs if you try to manipulate protected SOAP headers.

About this task

The client application sets properties on the Stub or Call object to send and receive implicit SOAP headers.

Procedure

1. Create a java.util.HashMap object
2. Add an entry to the HashMap object for each implicit SOAP header that the client wants to receive. The HashMap entry key is the QName of the SOAP header. The HashMap entry value is `null`.
3. Set the HashMap entry on the Stub or Call object. The property name is com.ibm.websphere.webservices.Constants.RESPONSE_SOAP_HEADERS. The value of the property is the HashMap.
4. Issue remote method calls against the Stub or Call object. The web services engine extracts the specified response headers from the web services response message and inserts them into the HashMap. After the remote method returns, the response headers are accessible from the HashMap object.

   A `JAXRPCException` error can occur if any of the following are true:
   - The HashMap contains a key that is not a QName.
   - The HashMap contains a key that represents a SOAP header that is declared protected by the owning component.

Results

You have a JAX-RPC web services client that can receive values from implicit SOAP headers.

Sending transport headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to send application-defined information along with your Web services requests by using transport headers.

Before you begin

You need a JAX-RPC web services client that you can enable to send transport headers.

Sending transport headers is supported only by web services clients, and only supported for the HTTP and JMS transports. The web services client must call the JAX-RPC APIs directly and not through any intermediary layers, such as a gateway function. Sending and retrieving transport headers on the web services server is done through non-web services APIs.

About this task

When using the JAX-RPC programming model, the client must set a property on the Stub or Call object to send values in transport headers. After you set the property, the values are set in all the requests for subsequent remote method invocations against that Stub or Call object until the associated property is set to `null` or the Stub or Call object is discarded.

To send values in the transport headers on outbound requests, modify the client code as follows:

Procedure

1. Create a java.util.HashMap object that contains the transport header identifiers.
2. Add an entry to the HashMap object for each transport header that you want the client to send.

a. Set the HashMap entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made as to the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class.

b. Set the HashMap entry value to a string that contains the value of the transport header.

3. Set the HashMap entry on the Stub or Call object using the `com.ibm.websphere.webservices.Constants.REQUEST_TRANSPORT_PROPERTIES` property. When the REQUEST_TRANSPORT_PROPERTIES property value is set, that HashMap is used on subsequent invocations to set the header values in the outgoing requests. If the REQUEST_TRANSPORT_PROPERTIES property value is set to `null`, no HashMap is used on subsequent invocations to set header values in outgoing requests. To learn more about these properties, see the transport header properties documentation.
4. Issue remote method calls against the Stub or Call object. The headers and the associated values from the HashMap are added to the outgoing request for each method invocation. If the invocation uses HTTP, then the transport headers are sent as HTTP headers within the HTTP request. If the invocation uses JMS, then the transport headers are sent as JMS message properties.

   If the property is not set correctly, you might experience API usage errors that result in a JAXRPCException error. The following requirements must be met, or the process fails:

   - The property value that is set on the Stub or Call object must be a HashMap object or `null`.
   - The HashMap must not be empty.
   - Each key in the HashMap must be a String object.
   - Each value in the HashMap must be a String object.

Results

You have a JAX-RPC web services client that is configured to send transport headers.

Retrieving transport headers with JAX-RPC

You can enable an existing Java API for XML-based RPC (JAX-RPC) Web services client to retrieve values from transport headers. For a request that uses HTTP, the transport headers are retrieved from HTTP headers found in the HTTP response message. For a request that uses Java Message Service (JMS), the transport headers are retrieved from the JMS message properties found on the JMS response message.

Before you begin

You need a web services client that you can enable to retrieve transport headers.

Retrieving transport headers is supported only by web services clients, and only supported for the HTTP and JMS transports. The web services client must call the JAX-RPC APIs directly and not through any intermediary layers, such as a gateway function. Sending and retrieving transport headers on the web services server is done through non-web services APIs.

About this task

When using the JAX-RPC programming model, the client must set a property on the Stub or Call object in order to retrieve values from the transport headers. After you set the property, values are read from responses for the subsequent method invocations against that Stub or Call instance until the associated property is set to `null` or the Stub or Call object is discarded.

To retrieve values from the transport headers on inbound responses, modify the client code.

Procedure

1. Create a java.util.HashMap object that contains the names of the transport headers to be retrieved from incoming response messages.
2. Add an entry to the HashMap for each header that you want to retrieve a value from every incoming response message.
   a. Set the HashMap entry key to a string that exactly matches the transport header identifier. You can define the header identifier with a reserved header name, such as Cookie in the case of HTTP, or the header identifier can be user-defined, such as `MyTransportHeader`. Certain header identifiers are processed in a unique manner, but no other checks are made to confirm the header identifier value. To learn more about the HTTP header identifiers that have unique consideration, read about transport header properties best practices. You can find common header identifier string constants, such as HTTP_HEADER_SET_COOKIE in the com.ibm.websphere.webservices.Constants class. The HashMap entry value is ignored and does not need to be set. An empty HashMap, for example, one that is non-null, but does not contain any keys, causes all the transport headers in the response to be retrieved.
3. Set the HashMap entry on the Stub or Call object using the com.ibm.websphere.webservices.Constants.RESPONSE_TRANSPORT_PROPERTIES property. When the HashMap is set, the RESPONSE_TRANSPORT_PROPERTIES property is used in subsequent invocations to retrieve the headers from the responses. If you set the property to `null`, no headers are retrieved from the response. To learn more about these properties, see the transport header properties documentation.
4. Issue remote method calls against the Stub or Call object. The values from the specified transport headers are retrieved from the response message and placed in the HashMap.

   If the property is not set correctly, you might experience API usage errors that result in a JAXRPCException error. The following requirements must be met, or the process fails:
   - The property value that is set on the Stub or Call object must be either `null` or an instance of a HashMap.
   - All the HashMap keys must be a string data type, and the keys must not be null.

Results

You have a JAX-RPC web service that can receive transport headers from incoming response messages.

Assembling web services applications

Assembling web services applications

You can assemble Java-based web services applications using assembly tools.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with the application server.

About this task

After you develop your web service application, you are now ready to assemble the application. Assembling a web service application consists of creating the Java Platform, Enterprise Edition (Java EE) modules that you can deploy onto application servers. The modules are created from code artifacts such as web application archives (WAR) files for JavaBeans applications or enterprise beans Java archive (JAR) files for enterprise beans applications. This packaging and configuring of code artifacts into enterprise application modules (EAR files) or standalone web modules is necessary for deploying the modules onto an application server.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. Assemble your web services enabled bean into the appropriate module.
   - For JavaBeans enabled as web services:
     a. “Assembling a WAR file that is enabled for web services from Java code” on page 1294.
     b. “Assembling a web services-enabled WAR file from a WSDL file” on page 1295.
   - For enterprise beans enabled as web services:
     a. “Assembling a JAR file that is enabled for web services from an enterprise bean.”
     b. “Assembling a web services-enabled enterprise bean JAR file from a WSDL file” on page 1292.

   Note: This product supports packaging enterprise beans in WAR files. If you include a web services-enabled enterprise bean JAR file into a WAR file, you must merge any information in the webservices.xml deployment descriptor files that are in the JAR files into the webservices.xml deployment descriptor in the WEB-INF directory of the WAR file. To learn more, see the EJB content in WAR modules information.

   Note: When developing faults for a JAX-WS application, it is a best practice to always include the fault bean that is generated by the JAX-WS tooling in the packaging of your JAX-WS application. However if your application does not use the fault bean classes that are generated by the JAX-WS tooling (that is, you use a bottom-up development approach starting from Java and you choose not to package the fault bean classes), the application server runtime environment dynamically generates the fault beans. Even so, it is a best practice to always package the fault bean.
3. Assemble the web services enabled module into an enterprise archive (EAR) file.
   - “Assembling a web services-enabled WAR into an EAR file” on page 1297.
   - “Assembling an enterprise bean JAR file into an EAR file” on page 1296.
4. Enable the EAR file for EJB modules that contain web services. When the EAR file contains Enterprise JavaBeans (EJB) modules that contain web services, you must run the **endptEnabler** command-line tool or an assembly tool before deployment to produce a web services endpoint WAR file. This tool is also used to specify whether the web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.
5. Assemble a web services-enabled WAR file into an EAR file.

Results

You have a web services-enabled EAR file that you can deploy onto the application server.

What to do next

Now you need to deploy the web services-enabled EAR file onto your application server. To learn more, read about deploying web services applications onto application servers

Assembling a JAR file that is enabled for web services from an enterprise bean

You can assemble a web service-enabled enterprise bean Java archive (JAR) file with an assembly tool using artifacts generated from tooling.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

You need the following artifacts that are generated from the **WSDL2Java** command-line tool to complete this task:
- An assembled enterprise bean JAR file that is not enabled for web services
- A compiled Java class for the service endpoint interface
- A Web Services Description Language (WSDL) file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptor, and Java API for XML-based remote procedure call (JAX-RPC) mapping file.

About this task

Assemble a web services-enabled enterprise bean JAR file from Java code by following the actions in the steps for this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.

Results

You have the artifacts required to web service-enable an Enterprise JavaBeans (EJB) module for web services. The artifacts are added to the JAR file. Now you need to configure the deployment descriptors so that you can deploy the web service into the application server run time environment.

Example

The `AddressBook.jar` JAR file contains the following files after assembly. The files added in this task are in bold. These files include the WSDL file, the deployment descriptors, and the JAX-RPC mapping file.

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookBean.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

What to do next

Assemble the EAR file so that you can deploy the EAR file into WebSphere Application Server.

Assembling a web services-enabled enterprise bean JAR file from a WSDL file

You can assemble a web services-enabled enterprise bean Java archive (JAR) file from a Web Services Description Language (WSDL) file with an assembly tool.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

You need the following artifacts to complete this task:
- An assembled enterprise bean JAR file that contains the Enterprise JavaBeans (EJB) implementation and all classes that generate from the **WSDL2Java** command-line tool when the role argument is `develop-server` and the container argument is `EJB`.
- A WSDL file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file.

About this task

Assemble a web services-enabled enterprise bean JAR file from a WSDL file by following the actions in the steps for this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.

Results

You have the artifacts required to web service-enable an EJB module for web services. The artifacts are added to the JAR file. Now you need to configure the deployment descriptors so that you can deploy the web service into the application server runtime environment.

Example

The `AddressBook.jar` JAR file contains the following files after assembly. The files added in this task are in bold. These files include the WSDL file, the deployment descriptors, and the JAX-RPC mapping file.

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookSoapBindingImpl.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

What to do next

For JAX-RPC web services, configure the `webservices.xml` deployment descriptor . You need to configure the deployment descriptors for the web service so that WebSphere Application Server can process the incoming web services requests.

Assembling a WAR file that is enabled for web services from Java code

You can assemble a web application archive (WAR) file that is enabled for web services from Java code with an assembly tool.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

For Java API for XML-Based Web Services (JAX-WS) web service applications, you need the portable artifacts that are generated by the **wsgen** command-line tool when starting from a service endpoint implementation to complete this task. The **wsgen** tool processes a compiled service endpoint implementation class as input and generates the following portable artifacts:

- any additional Java Architecture for XML Binding (JAXB) classes that are required to marshal and unmarshal the message contents. The additional classes include classes that are represented by the @RequestWrapper annotation and the @ResponseWrapper annotation for a wrapped method.
- a WSDL file if the optional *-wsdl* argument is specified. The **wsgen** command does not automatically generate the WSDL file. The WSDL file is automatically generated when you deploy the service endpoint.

For Java API for XML-based RPC (JAX-RPC) web service applications, you need the following artifacts that are generated by the **WSDL2Java** command-line tool to complete this task:

- An assembled WAR file that contains the `web.xml` file, but is not enabled for web services.
- The Java class for the service endpoint interface
- A Web Services Description Language (WSDL) file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file classes that are generated by the **WSDL2Java** command.

About this task

Assemble a web services-enabled WAR file from Java code by following the actions in the steps for this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Import the JavaBeans implementation and the artifacts generated by the command-line tooling into the assembly tool.
4. Migrate WAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.

Results

The artifacts required to enable the web module for web services are added to the WAR file.

What to do next

Now you can assemble the WAR file that is enabled for Web services into an EAR file. To learn more, read about assembling a web services-enabled WAR into an EAR file.

Assembling a web services-enabled WAR file from a WSDL file

You can assemble a web application archive (WAR) file from a Web Services Description Language (WSDL) file that is enabled for web services.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

For Java API for XML-Based Web Services (JAX-WS) web service applications, you need the portable artifacts that are generated by the **wsimport** command-line tool when starting from a WSDL file to complete this task. The **wsimport** tool processes a WSDL file as input and generates the following portable artifacts:

- Service Endpoint Interface (SEI)
- Service class
- Exception classes that are mapped from the `wsdl:fault` class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

You can package the generated artifacts in a web application archive (WAR) file with the WSDL file and schema documents along with the endpoint implementation that you plan to deploy.

For Java API for XML-based RPC (JAX-RPC) web service applications, you need the following artifacts that are generated by the **WSDL2Java** command-line tool to complete this task:

- An assembled WAR file that contains the Enterprise JavaBeans (EJB) implementation, all the classes that generate from the **WSDL2Java** command-line tool and the `web.xml` deployment descriptor file.
- A WSDL file
- The complete `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file.

About this task

Assemble a web services-enabled WAR file from a WSDL file by following the actions in the steps for this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Import the JavaBeans implementation and the artifacts generated by the command-line tooling into the assembly tool.
4. Migrate JAR files created with the Assembly Toolkit, Application Assembly Tool or a different tool to the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer information.

Results

The artifacts required to enable the web module for web services is added to the WAR file.

What to do next

Now you can assemble the WAR file that is enabled for Web services into an EAR file. To learn more, read about assembling a web services-enabled WAR into an EAR file.

Assembling an enterprise bean JAR file into an EAR file

You can assemble an enterprise bean Java archive (JAR) file into an enterprise archive (EAR) file with an assembly tool. Assembling the JAR file, and now the EAR file, are required tasks to enable Java code for web services.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

Before assembling a web services-enabled EAR file you must assemble an enterprise bean JAR file that you want to enable for Web services. To learn more about the artifacts that are needed for the assembly of the enterprise bean JAR file, see the information on assembling an enterprise bean JAR file from Java code that is enabled for web services.

Restriction: Do not include a pound sign (#) in the name of files that are packaged within an application archive. Due to internal processing, the application server fails to correctly deploy the application when a pound sign is included in a file name within the application archive. When this failure occurs, an exception might occur when the application is being processed. Also, parts of the application might be missing after the application is deployed. To address this issue, rename any file names within the application archive so that they do not contain a pound sign.

About this task

To assemble a web services-enabled EAR file:

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Assemble the web services-enabled JAR file into an EAR file. The EAR file can contain an enterprise bean or application client JAR files, WAR files, web applications, and metadata describing the applications or `application.xml` files.

Results

A web services-enabled EAR file.

Example

In the following example, there is an `application.xml` deployment descriptor packaged with a web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.jar
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
 <application id="Application_ID">
  <display-name>AddressBookJ2WEE</display-name>
  <description>AddressBook EJB Example from Java</description>
  <module id="EjbModule_1">
   <ejb>AddressBook.jar</ejb>
  </module>
 </application>
```

What to do next

You can enable an EAR file for EJB modules that contain web services. Then, deploy the EAR file into WebSphere Application Server.

Assembling a web services-enabled WAR into an EAR file

You can assemble a web services-enabled web application archive (WAR) file into an enterprise archive (EAR) file with an assembly tool.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

Restriction: Do not include a pound sign (#) in the name of files that are packaged within an application archive. Due to internal processing, the application server fails to correctly deploy the application when a pound sign is included in a file name within the application archive. When this failure occurs, an exception might occur when the application is being processed. Also, parts of the application might be missing after the application is deployed. To address this issue, rename any file names within the application archive so that they do not contain a pound sign.

About this task

Assemble a web services-enabled WAR file into an EAR file using the steps provided in this task section.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in theRational Application Developer documentation.
2. Assemble the web services-enabled WAR file into an EAR file. Assemble the EAR file that contains the JAR or WAR files. The EAR file can contain an enterprise bean or application client JAR files; web applications or WAR files; and metadata describing the applications or `application.xml` files. To learn more about how to assemble the WAR file, see the assembling applications documentation.

Results

A web services-enabled EAR file.

Example

In the following example, there is an `application.xml` deployment descriptor packaged with a web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.war
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
"http://java.sun.com/dtd/application_1_3.dtd">
 <application id="Application_ID">
  <display-name>AddressBook</display-name>
  <description>AddressBook Example from Java bean</description>
  <module id="WebModule_1">
   <web>
    <web-uri>AddressBook.war</web-uri>
    <context-root>/AddressBook</context-root>
   </web>
  </module>
 </application>
```

What to do next

Deploy your web services.

Enabling an EAR file for EJB modules that contain web services

When your enterprise archive (EAR) file contains enterprise JavaBeans (EJB) modules that contain Web services, you must run the **endptEnabler** command-line tool or an assembly tool before deployment to produce a web services endpoint web application archive (WAR) fiile.

Before you begin

Assemble an enterprise Java archive (JAR) file that is enabled for web services from an enterprise bean. The enterprise JAR file is an artifact that is required to build the EAR file.

About this task

You can add router modules to your application that is enabled for web services with either the **endptEnabler** command-line tool or with assembly tools provided with WebSphere Application Server. The tool that you choose to use for this task depends on your preference to work with a command-line tool or a graphical user interface. See the assembly tools documentation to learn how to use assembly tool to accomplish this task.

These tools add one or more router modules to the EAR file for each web service-enabled enterprise JavaBeans (EJB) module contained in the EAR file. A router module provides an endpoint for the web service in a particular EJB module.

You should not modify the contents of the EJB module or the web module that was generated using the **endptEnabler**command-line tool. If you do, an error occurs during run time. The following is an example of the error that displays:

```
"Error]- WSWS3142E: Error: Could not find web services engine.]: javax.servlet.ServletException: WSWS3142E:
Error: Could not find web services engine."
```

Each router module supports a specific transport such as HTTP or Java Message Service (JMS). If no enterprise bean JAR modules are located in the EAR file, it is not necessary to use these tools.

Procedure

Enable an EAR file with the **endptEnabler** command-line tool. In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application.

What to do next

Deploy the EAR file into WebSphere Application Server. An assembled EAR file that is enabled for web services is required for deployment.

Enabling an EAR file for web services with the endptEnabler command:

Use the **endptEnabler** command-line tool to enable an enterprise archive (EAR) file for Enterprise JavaBeans (EJB) modules that contain web services and to specify whether the web services are exposed using SOAP over Java Message Service (JMS) or SOAP over HTTP.

Before you begin

Before doing this task, you need to assemble a web services-enabled enterprise Java archive (JAR) into an EAR file.

About this task

The **endptEnabler** command-line tool adds one or more router modules to the EAR file for each Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) based Web service-enabled enterprise bean Java archive (JAR) module within the EAR file. A router module provides an endpoint for the web services in a particular enterprise bean JAR module.

Each router module supports a specific transport such as HTTP or JMS. An HTTP router module is a web application archive (WAR) module that provides an HTTP endpoint for each of the web services contained within a particular enterprise bean JAR module. Likewise, a JMS router module is an enterprise bean JAR module that contains a Message Driven Bean (MDB) that serves as the message listener for requests intended for the web service endpoints.

If no enterprise bean JAR modules exist in the EAR file, it is not necessary to use this tool.

Procedure

1. Invoke the **endptEnabler** command.

   Invoke the **endptEnabler** command from the *install_root*/bin directory.
2. Enter the name of the EAR file, when prompted.
3. Enter various input values as requested by the **endptEnabler** command. You are prompted for various input values for each enterprise bean JAR module that is enabled for web services in the EAR file. Typically, you accept the defaults for each prompt. To learn about the properties of this command, see the **endptEnabler** command documentation.
   a. Specify an HTTP router module to transport your EJB-based web service. Use the `-transport http` option to indicate the web service is available using HTTP. One router module is created for each enterprise bean JAR file that contains either JAX-WS or JAX-RPC web services.
   b. Specify an JMS router module to transport your EJB-based web service. Use the `-transport jms` option to indicate the web service is available using JMS. One router module is created for each enterprise bean JAR file that contains either JAX-WS or JAX-RPC web services.

Results

An HTTP or JMS router module is added to the EAR file for each enterprise bean JAR module within the EAR file that contains web services endpoints. For HTTP, a context-root is configured for the application so that the web service can be invoked through a Web address. The web address used to invoke the web service is:

`http://host[:port]/<context-root>/services/<port-component-name>`

Ensure that you install the HTTP or Java Message Service (JMS) router module that you generated with the **endptEnabler** command onto the same target as your web services enterprise bean JAR files. These HTTP or JMS router modules are included in your web services application and they need to use the runtime libraries of the application server.

What to do next

Deploy the EAR file onto your application server. An assembled EAR file that is enabled for web services is required for deployment.

If you are using JMS as a transport for your web service requests, define the various JMS objects such as queues, topics, or connection factories, that will be used by your application prior to installing the application.

endptEnabler command:

The **endptEnabler** command is used to enable a set of web services within an enterprise archive (EAR) file. The **endptEnabler** command must run on EAR files containing Enterprise JavaBeans (EJB) modules that are enabled for web services.

Each router module provides a web service endpoint for a particular transport. For example, you can add a HTTP router module so that the web service can receive requests over the HTTP transport. Or, you can add a Java Message Service (JMS) router module so that the web service can receive requests from a JMS queue or topic.

In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application. The **endptEnabler** command makes a backup copy of your original EAR file in the event that you need to remove or add services at a later time. If your EAR file contains an enterprise bean Java archive (JAR) file that is enabled for web services, you must run the **endptEnabler** command before the EAR file is deployed. Otherwise, you do not need to run the command.

endptEnabler usage syntax

Invoke the **endptEnabler** command from the WebSphere Application Server `bin` directory. The command syntax is presented in the following example:

```
endptEnabler
    [-verbose|-v]
    [-quiet|-q]
    [-help|-h|-?]
    [-properties|-p properties-filename]
    [-transport|-t default-transports]
    [-enableHttpRouterSecurity]
    [-enableClientCertSecurity]
    [ear-filename]
    [-classpath|-cp]
```

All parameters are optional and described in the following list:

-verbose, -v
This parameter details and displays progress messages as the endptEnabler tool processes the EAR file. This command-line option is mapped to the verbose global property.

-quiet, -q
This parameter makes sure that there are no displays of per-module progress messages as the endptEnabler tool processes the EAR file. This command-line option is mapped to the quiet global property.

-help, -h, -?
This parameter displays a brief help message that explains the various options.

-properties, -p <properties-filename>
This parameter reads properties from the *properties-filename* properties and controls the behavior of the endptEnabler tool.

-transport, -t <default-transports>
This parameter specifies the default list of transports for which router modules are created for each enterprise bean JAR file contained in the EAR file. This command-line option is mapped to the *defaultTransports* global property. The following are examples of this parameter:

```
-transport http (the default)
-transport jms
-t http,jms
```

-enableHttpRouterSecurity
This parameter enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB modules are secured in the enterprise bean JAR file. This command-line option is mapped to the `http.enableRouterSecurity` global property.

-enableClientCertSecurity
This parameter enables you to add the following login-config stanza to the `web.xml` file for certificate authentication, instead of basic-auth or form-login. This command-line option is mapped to the `http.enableClientCertSecurity` global property.

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

Note: Use the -enableClientCertSecurity parameter with the -enableHttpRouterSecurity parameter.

<ear-filename>
This parameter specifies the name of the EAR file to be processed.

If the **ear-filename** parameter is not entered on the command line, the interactive mode is used. In the interactive mode, you are prompted for the EAR file name, the router module names and other important values as the processing occurs. The following dialog is an example of the endptEnabler interactive mode.

In this dialog, the user input is in fixed width font and the endptEnabler output is in bold.

```
endptEnabler<enter>
WSWS2004I: IBM WebSphere Application Server Release 5
WSWS2005I: Web Services Enterprise Archive Endpoint Enabler Tool.
WSWS2007I: (C) COPYRIGHT International Business Machines Corp. 1997, 2003
WSWS2006I: Please enter the name of your EAR file: AddressBook.ear<enter>

WSWS2003I: Backing up EAR file to: AddressBook.ear~

WSWS2016I: Loading EAR file: AddressBook.ear
WSWS2017I: Found EJB Module: AddressBookEJB.jar

WSWS2029I: Enter http router name for EJB Module AddressBookEJB
[AddressBookEJB_HTTPRouter.war]:<enter>
WSWS2030I: Enter http context root for EJB Module AddressBookEJB
[/AddressBookEJB]:<enter>
WSWS2024I: Adding http router for EJB Module AddressBookEJB.jar.
WSWS2036I: Saving EAR file AddressBook.ear...
WSWS2037I: Finished saving the EAR file.
WSWS2018I: Finished processing EAR file AddressBook.ear.
```

If the **ear-filename** parameter is entered on the command-line, the non-interactive mode is used. In the non-interactive mode, the router module names and other important values are determined from the user-specified properties or default values.

-classpath, -cp
This parameter enables you to specify separate JAR files in the class path for the **endptEnabler** command. Use this parameter if the EJB file references Java types in separate JAR files that do not exist within the EAR file. You can specify multiple values for this parameter. For example:

```
endptEnabler -classpath C:\MyWork\Utility.jar;D:\SharedFiles\Hello.zip;HelloWorld.jar
```

When you do not specify this parameter, you might encounter an error that is similar to the following message:

```
WSWS2021I: Skipping the enterprise bean module module_name
because it contains no web services.
```

endptEnabler properties

With the **endptEnabler** command you can control its runtime behavior by specifying a set of properties with the -properties command-line option. These properties are organized in one of two ways: global and per-module. Global properties affect the overall behavior of the tool as it processes multiple enterprise bean JAR modules within the EAR file. Per-module properties affect the processing of a particular enterprise bean JAR module.

Table 136. Global properties for the **endptEnabler** *command. Use these global properties of the endptEnabler command when enabling a set of web services within an enterprise archive (EAR) file.*

Property name	Description	Default value
verbose	Displays detailed progress messages.	False
quiet	Displays only brief progress messages.	False
http.enableRouterSecurity	Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB modules are secured in the enterprise bean JAR file.	False

http.enableClientCertSecurity	Enables you to add the following login-config stanza to the `web.xml` file for certificate authentication, instead of basic-auth or form-login:
	`<login-config>`
	`  <auth-method>CLIENT-CERT</auth-method>`
	`</login-config>`
		False

http.routerModuleNameSuffix	Specifies the suffix used to construct default HTTP router module names. The `.war` extension is added by the **endptEnabler** command.	_HTTPRouter
jms.routerModuleNameSuffix	Specifies the suffix used to construct default JMS router module names. The `.jar` extension is added by the **endptEnabler** command.	_JMSRouter
jms.defaultDestinationType	Specifies the default destination type to use for all JMS router modules that are added to the EAR file. This type is either queue or topic.	queue
defaultTransports	Specifies the default list of transports for which router modules are created. The list can contain the values `http` and `jms`. Multiple values are separated by a comma. Examples are: `http, jms` and `http,jms`.	http

The following table describes the per-module properties supported by the **endptEnabler** command. The *ejbJarName* variable refers to the name of an enterprise bean JAR module within the EAR file, without the `.jar` extension.

Table 137. Per-module properties for the **endptEnabler** *command. Use these properties of the* **endptEnabler** *command that affect the processing of a particular enterprise bean JAR module.*

Property name	Description	Default value
`<ejbJarName>.transports`	Lists the transports for which router modules are created for a particular enterprise bean JAR file. The list can contain the values `http` and `jms`. Multiple values are separated by a comma. Examples are: `http, jms` and `http,jms`.	http
`<ejbJarName>.http.skip`	Specifies the flag which bypasses the addition of an HTTP router module, even if it otherwise is added based on other properties. Valid values are `true` and `false`.	false
`<ejbJarName>` `.http.routerModuleName`	Specifies the name of the HTTP router module for a particular enterprise bean JAR file.	*ejbJarName*_HTTPRouter
`<ejbJarName>.http.contextRoot`	Specifies the context root associated with the HTTP router module for a particular enterprise bean JAR file.	/*ejbJarName*
`<ejbJarName>.jms.skip`	Specifies the flag that bypasses the addition of an JMS router module even if it otherwise is added based on other properties. Valid values are `true` and `false`.	false
`<ejbJarName>.jms.` `routerModuleName`	Specifies the name of the JMS router module for a particular enterprise bean JAR file.	*ejbJarName*_JMSRouter
`<ejbJarName>.jms.` `activationSpecJndiName`	Specifies the Java Naming and Directory Interface (JNDI) name of the activation specification that is configured for the Message Driven Bean (MDB) within the JMS router module.	null

Table 137. Per-module properties for the **endptEnabler** *command (continued). Use these properties of the* **endptEnabler** *command that affect the processing of a particular enterprise bean JAR module.*

Property name	Description	Default value
<ejbJarName>.jms. listenerInputPortName	Specifies the name of the listener port to configure for the MDB within the JMS router module. The listener port is configured only if an activationSpecJndiName property is not specified.	null
<ejbJarName>.jms.destinationType	Specifies the JMS destination type associated with the MDB within the JMS router. Valid values are queue and topic.	queue
<ejbJarName>.<port_local_name> .http.urlPattern= <url_pattern_name>	Specifies the URL pattern for ports. If you have EJB module with the indicated name that has a port with the indicated local name, you can specify the HTTP URL pattern with this property. This property only applies to HTTP router modules. It has no affect on JMS router modules.	null

Properties example

Suppose an EAR file contains an enterprise bean JAR file named, StockQuoteEJB.jar that contains web services. The following set of properties can be used to control the **endptEnabler** command runtime behavior as it processes the EAR file:

```
StockQuoteEJB.transports=http,jms

StockQuoteEJB.http.routerModuleName=StockQuoteEJB_HTTP

StockQuoteEJB.http.contextRoot=/StockQuote

StockQuoteEJB.jms.routerModuleName=StockQuoteEJB_JMS

StockQuoteEJB.jms.destinationType=queue
```

endptEnabler examples

The following commands are examples of how the **endptEnabler** command can be used:

```
endptEnabler MyApp.ear

endptEnabler -t jms,http MyApp.ear

endptEnabler -v -properties MyApp.props MyApp.ear

endptEnabler -q -t jms MyApp.ear

endptEnabler -v -t http,jms
```

Assembling web services-enabled clients

Assembling a web services-enabled client JAR file into an EAR file

Now that you have generated your application artifacts, you need to assemble these artifacts to create an enterprise archive (EAR) file that is used in the web services application.

Before you begin

For Java API for XML-Based Web Services (JAX-WS) web service applications, you need the portable artifacts that are generated by the **wsimport** command-line tool when starting from a WSDL file to complete this task. The **wsimport** tool processes a WSDL file as input and generates the following portable artifacts:

- Service Endpoint Interface (SEI)
- Service class
- Exception classes that are mapped from the wsdl:fault class (if any)
- Java Architecture for XML Binding (JAXB) generated type values which are Java classes mapped from XML schema types

- An assembled client module that contains the implementation, all of the classes generated by the **wsimport** command-line tool and the `ejb-jar.xml` deployment descriptor or the `application-client.xml` deployment descriptor. This module can be:
  - An application client module that contains the `META-INF/application-client.xml` file.
  - An Enterprise JavaBeans (EJB) module that contains the `META-INF/ejb-jar.xml` file.

For Java API for XML-based RPC (JAX-RPC) web service applications, you need the following artifacts that are generated from the **WSDL2Java** command-line tool to complete this task:
- An assembled client module that contains the implementation, all of the classes generated by the **WSDL2Java** command-line tool and the `ejb-jar.xml` deployment descriptor or the `application-client.xml` deployment descriptor. This module can be:
  - An application client module that contains the `META-INF/application-client.xml` file.
  - An Enterprise JavaBeans (EJB) module that contains the `META-INF/ejb-jar.xml` file.
- The WSDL file that you used to develop the client.
- The templates for the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` deployment descriptor, if used.
- A generated Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptor.

Restriction: Do not include a pound sign (#) in the name of files that are packaged within an application archive. Due to internal processing, the application server fails to correctly deploy the application when a pound sign is included in a file name within the application archive. When this failure occurs, an exception might occur when the application is being processed. Also, parts of the application might be missing after the application is deployed. To address this issue, rename any file names within the application archive so that they do not contain a pound sign.

About this task

You can use assembly tools included with WebSphere Application Server to assemble web services-enabled client applications.

Assemble the client code and artifacts that enable the application client to access a web service with steps provided:

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Import the client implementation and the artifacts generated by the command-line tooling into the assembly tool.
4. Migrate JAR files created with the Rational Application Developer assembly tool. To migrate files, import your JAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
5. Assemble the JAR file into an enterprise archive (EAR) file using typical assembly techniques if the client runs in a container.

Results

You have assembled the artifacts required to enable the client application for web services into an EAR file.

Example

This example of the assembly process uses the `AddressBookClient.jar` JAR file the `AddressBookClient.ear` EAR file:

```
META-INF/MANIFEST.MF
META-INF/application-client.xml
META-INF/wsdl/AddressBook.wsdl
META-INF/AddressBook_mapping.xml

com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookClient.jar` file into the `AddressBookClient.ear` file, the `AddressBookClient.ear` file contains the following files:

```
META-INF/MANIFEST.MF
AddressBookClient.jar
META-INF/application.xml
```

What to do next

For Java API for XML-Based Web Services (JAX-WS) applications, you are ready to deploy the web services client application.

For Java API for XML-based RPC (JAX-RPC) applications, you need to configure the client deployment descriptor bindings with an assembly tool so that the client can communicate with a web service that is deployed on a server.

Assembling a web services-enabled client WAR file into an EAR file

Now that you have generated your application artifacts, you need to assemble these artifacts to create an enterprise archive (EAR) file that is used in the web services application.

Before you begin

You can assemble Java-based web services modules with assembly tools provided with WebSphere Application Server.

Restriction: Do not include a pound sign (#) in the name of files that are packaged within an application archive. Due to internal processing, the application server fails to correctly deploy the application when a pound sign is included in a file name within the application archive. When this failure occurs, an exception might occur when the application is being processed. Also, parts of the application might be missing after the application is deployed. To address this issue, rename any file names within the application archive so that they do not contain a pound sign.

About this task

Assemble the client code and artifacts that enable the application client to access a web service with steps provided:

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.

3. Migrate WAR files created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to an assembly tool. Read about importing web application archive (WAR) files using an assembly tool in the Rational Application Developer documentation.

Results

You have assembled the artifacts required to enable the client application for web services into an EAR file.

Example

This example of the assembly process uses the `AddressBookWeb.war` WAR file and the `AddressBook.ear` EAR file:

```
WEB-INF/MANIFEST.MF
WEB-INF/web.xml
WEB-INF/wsdl/AddressBook.wsdl
WEB-INF/AddressBook_mapping.xml
WEB-INF/ibm-webservicesclient-ext.xmi (optional)
WEB-INF/ibm-webservicesclient-bnd.xmi
com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookWeb.war` file into the `AddressBook.ear` file, the `AddressBook.ear` file contains the following files:

```
META-INF/MANIFEST.MF
AddressBookWeb.war
META-INF/application.xml
```

What to do next

For Java API for XML-Based Web Services (JAX-WS) applications, you are ready to deploy the web services client application.

For Java API for XML-based RPC (JAX-RPC) applications, you need to configure the client deployment descriptor bindings with an assembly tool so that the client can communicate with a web service that is deployed on a server.

Chapter 29. Developing web services - Addressing (WS-Addressing)

The Web Services Addressing (WS-Addressing) support in this product provides the environment for web services that use the World Wide Web Consortium (W3C) WS-Addressing specifications. This family of specifications provide transport-neutral mechanisms to address web services and to facilitate end-to-end addressing.

Using the Web Services Addressing APIs: Creating an application that uses endpoint references

This product provides application programming interfaces for applications that have to create endpoint references and use those endpoint references to target web service endpoints.

Before you begin

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Complete this task if you are a web service developer who needs to create endpoint references within an application, and then use these references to target web service resource instances. For example, a WSRF application developer.

Procedure

1. Create a web service that is referenced by an endpoint reference, and a client that accesses the web service. For JAX-WS applications, use the instructions in “Creating a JAX-WS web service application that uses Web Services Addressing.” For JAX-RPC applications, use the instructions in “Creating a JAX-RPC web service application that uses Web Services Addressing” on page 1312
2. Optional: You can extend the application that you created in the previous step so that it conforms to the Web Services Resource Framework (WSRF) specifications, by following the instructions in Creating stateful web services by using the Web Services Resource Framework.

Creating a JAX-WS web service application that uses Web Services Addressing

Web Services Addressing (WS-Addressing) aids interoperability between web services by defining a standard way to address web services and provide addressing information in messages. This task describes the steps that are required to create a JAX-WS web service that is accessed using a WS-Addressing endpoint reference. The task also describes the extra steps that are required to use stateful resources as part of the web service.

Before you begin

The steps that are described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Complete this task if you are creating a JAX-WS web service that uses the WS-Addressing specification. This task uses the JAX-WS WS-Addressing APIs to create the required endpoint reference. Alternatively, you can create endpoint references by using the IBM proprietary WS-Addressing API, and convert them into JAX-WS API objects for use with the rest of the application.

Procedure

1. Provide a web service interface that returns an endpoint reference to the target service.

   The interface must return an endpoint reference, which it can do by using a factory operation or a separate factory service. The target service can front a resource instance, for example a shopping cart.
2. Implement the web service created in the previous step. For the WS-Addressing portion of the implementation, complete the following steps:
   a. Optional: Include annotations to specify WS-Addressing behavior. See “Web Services Addressing annotations” on page 1359 for more details.
   b. Optional: If your interface involves a web service that fronts a resource instance, create or look up the resource instance.
   c. Optional: If you are using a resource instance, obtain the identifier of the resource. The resource identifier is application dependent and might be generated during the creation of the resource instance.

      Attention: Do not put sensitive information in the resource identifier, because the identifier is propagated in the SOAP message.
   d. Create an endpoint reference that references the web service by following the instructions in “Creating endpoint references by using the JAX-WS Web Services Addressing API” on page 1310. If you are using a resource instance, pass in the resource identifier as a parameter.
   e. Return the endpoint reference.
3. If your web service uses resource instances, extend the implementation to match incoming messages to the appropriate resource instances. Because you associated the resource identifier with the endpoint reference that you created earlier, any incoming messages targeted at that endpoint reference contain the resource identifier information as a reference parameter in the SOAP header of the message. Because the resource identifier is passed in the SOAP header, you do not have to expose it on the web service interface. When WebSphere Application Server receives the message, it puts this information into the message context on the thread. Extend the implementation to undertake the following actions:
   a. Obtain the resource instance identifier from the message context.
      - If you are using the 2005/08 WS-Addressing namespace, use the REFERENCE_PARAMETERS property of the MessageContext class.
      - If you are using the 2004/08 WS-Addressing namespace, you must use the IBM WS-Addressing API, specifically the EndpointReferenceManager.getReferenceParameterFrom MessageContext(QName resource_id) method.

      Use the following method for the 2005/08 namespace:

```
...
List resourceIDList = (List)getContext().getMessageContext().get(MessageContext.REFERENCE_PARAMETERS);
...
```

      Use the following method for the 2004/08 namespace:

```
...

String resource_identifier =
        EndpointReferenceManager.getReferenceParameterFromMessageContext(PRINTER_ID_PARAM_QNAME);
...
```

   b. Forward the message to the appropriate resource instance.
4. Optional: Configure a proxy client to communicate with the service.
   a. Use the wsimport or xjc tool to generate the artifacts required by the client.

      Note: If you want to use the 2004/08 WS-Addressing specification, specify the provided binding file, *app_server_root*/util/SubmissionEndpointReference.xjb, as the -b parameter of the tool. This parameter tells the tool to generate endpoint reference objects by using the SubmissionEndpointReference class that is part of the IBM implementation of the standard JAX-WS API. If you do not specify this bindings file, the resulting endpoint reference objects will not work with the standard JAX-WS API.

b. In the client code, create an instance of the service class.

c. Obtain a proxy object from the service class. There are several ways to use the JAX-WS API to obtain proxy objects. For example, there are several getPort methods on the Service class and one on the EndpointReference class. For more information, refer to the API documentation.

d. Optional: Use the Addressing or SubmissionAddressing feature to enable WS-Addressing support. For example, create a proxy by using a getPort method that accepts web service features as a parameter. If you prefer, you can enable WS-Addressing support by using another method, such as policy sets. For more information see “Enabling Web Services Addressing support for JAX-WS applications” on page 1324.

e. Use the proxy object to invoke the service method that returns the endpoint reference.

The following sample code shows a client invoking a web service to add two numbers together. The web service issues a ticket (the resource identifier) to the client, and requires the client to use this ticket when invoking the web service.

The client creates two proxies. The first proxy obtains the ticket as an endpoint reference from the service. The second proxy uses the AddressingFeature class to enable WS-Addressing for the 2005/08 specification, and invokes the service to add the two numbers together.

```
...
CalculatorService service = new CalculatorService();
// Create the first proxy
Calculator port1 = service.getCalculatorServicePort();
// Obtain the ticket as an endpoint reference from the service
W3CEndpointReference epr = port1.getTicket();

// Create the second proxy, using an addressing feature to enable WS-Addressing
Calculator port2 = epr.getPort(Calculator.class, new AddressingFeature());
// Invoke the service to add the numbers
int answer = port2.add(value0, value1);
System.out.println("The answer is: " + answer);
...
```

Note: If the metadata of the endpoint reference conflicts with the information already associated with the outbound message, for example if the proxy object is configured to represent a different interface, a javax.xml.ws.WebServiceException exception is thrown on attempts to invoke the endpoint.

If you want to set message-addressing properties, such as a reply to endpoint, you must use the IBM proprietary WS-Addressing SPI and the BindingProvider class, as described in “Specifying and acquiring message-addressing properties by using the IBM proprietary Web Services Addressing SPIs” on page 1321.

5. Optional: Configure a Dispatch client to communicate with the service. You can configure a client in different ways; the following steps describe one example.

   a. Create an instance of the service.

   b. Add a port to the service object.

   c. Create an instance of the Dispatch class, passing in the endpoint reference.

   d. Create a Dispatch<T> object. Use the Service.createDispatch method with the following parameters:

      - The endpoint reference returned by the service, which represents the resource to forward messages to.
      - An array of web service features. Include one or more WS-Addressing features to enable WS-Addressing. See “Enabling Web Services Addressing support for JAX-WS applications” on page 1324 for more details.

      There are several variations of the Service.createDispatch method; see the API documentation for more details.

   e. Compose the client request message.

   f. Invoke the service endpoint with the Dispatch client.

   The following code shows an example fragment of a Dispatch client that enables 2004/08 WS-Addressing.

```
...
CalculatorService service = new CalculatorService();
Dispatch(<SOAPMessage> dispatch = service.createDispatch(
    endpointReference,
    SOAPMessage.class,
    Service.Mode.MESSAGE,
    new SubmissionAddressingFeature(true));
...
```

Results

The web service and client are configured to use endpoint references through the WS-Addressing support. For a detailed example that includes code, see “Example: Creating a web service that uses the JAX-WS Web Services Addressing API to access a generic web service resource instance” on page 1318.

What to do next

- Refer to “Web Services Addressing security” on page 1361 for information about security with WS-Addressing.
- Deploy the application. If you used WS-Addressing annotations or features in the code, you do not have to take any additional steps to enable WS-Addressing support. For more information and for other scenarios that might require additional steps, for example enabling WS-Addressing support by using policy sets, see “Enabling Web Services Addressing support for JAX-WS applications” on page 1324.

Creating endpoint references by using the JAX-WS Web Services Addressing API

Endpoint references are a primary concept of the Web Services Addressing (WS-Addressing) interoperability protocol, and provide a standard mechanism to encapsulate information about specific web service endpoints. This product provides interfaces for you to create endpoint references by using the standard JAX-WS API.

About this task

This task is a subtask of “Creating a JAX-WS web service application that uses Web Services Addressing” on page 1307.

Complete this task if you are writing an application that uses the standard JAX-WS WS-Addressing API. Such applications require endpoint references to target web service endpoints. The standard JAX-WS API is designed to create only simple endpoint references, and therefore has the following restrictions:

- You cannot create highly available or workload managed endpoint references.
- You cannot create endpoint references that represent stateful session beans.
- You cannot use classes that are created by using the JAX-WS API with the IBM proprietary WS-Addressing SPI.

You can overcome these restrictions by using the IBM proprietary WS-Addressing API to create the endpoint references and then converting them into standard JAX-WS endpoint references that can be used by the rest of the application.

Procedure

- If an endpoint needs to create an endpoint reference that represents itself, use the getEndpointReference method of the web service context object, passing in an Element object representing the reference parameters to be associated with the endpoint reference (or a null object if you do not want to specify any reference parameters).

  By default, this method creates a W3CEndpointReference object. If you want to create a SubmissionEndpointReference object, representing an endpoint that conforms to the 2004/08 WS-Addressing specification, pass the endpoint reference type as a parameter. For example, the following code fragment uses the getEndpointReference method to return a W3CEndpointRerence object that has a ticket ID associated with it:

```
...
@WebService(name="Calculator",
            targetNamespace="http://calculator.org")

public class Calculator {
    @Resource
    WebServiceContext wsc;

    ...
    // Create the ticket id
    element = document.createElementNS(
                      "http://calculator.jaxws.axis2.apache.org", "TicketId");
    element.appendChild( document.createTextNode("123456789") );
    ...

    public W3CEndpointReference getEPR() {
        // Get the endpoint reference and associate the ticket id
        // with it as a reference parameter
        W3CEndpointReference epr = (W3CEndpointReference)wsc.getEndpointReference(element);

        return epr;
    }
...
```

The following line of code shows how to create a 2004/08 endpoint reference for the preceding sample:

```
SubmissionEndpointReference epr = (SubmissionEndpointReference)
       wsc.getEndpointReference(SubmissionEndpointReference.class, element);
```

- If an endpoint needs to create an endpoint reference that represents a different endpoint, use either the W3CEndpointReferenceBuilder class or the SubmissionEndpointReferenceBuilder class, depending on the namespace that you want to use.
  1. Create an instance of the appropriate builder class. Use the W3CEndpointReferenceBuilder class if you want to create an endpoint reference that complies with the 2005/08 WS-Addressing specification. Use the SubmissionEndpointReferenceBuilder class if you want to create an endpoint reference that complies with the 2004/08 WS-Addressing specification.
  2. Set the following property or properties of the builder instance according to the location of the endpoint.
     - If the endpoint is in another module in this application, set the serviceName and endpointName properties to appropriate values. You must set the serviceName property before you set the endpointName property, otherwise the application throws an error. The endpoint reference that is returned contains a suitable address for the endpoint, as determined by the implementation.

       Note: This behavior differs from the IBM WS-Addressing API, in that creating an endpoint reference using the com.ibm.websphere.wsaddressing.EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method is not restricted to endpoints in the same application.
     - If the endpoint is in another Java EE application, set the address property to point to the endpoint.
  3. Optional: Set other properties of the builder instance as required. For example, if the web service is used to access a resource instance, use the referenceParameter property to associate the identifier of the resource with the endpoint reference. For more information on the properties that you can set, see the API documentation.
  4. Invoke the build method on the builder instance to obtain the endpoint reference.

  For example, the following code fragment uses the W3CEndpointReferenceBuilder class to obtain an endpoint reference that complies with the 2005/08 specification, and points to an endpoint that is in another application:

```
...
@WebService(name="Calculator", targetNamespace="http://calculator.org")
public class Calculator {

    public W3CEndpointReference getEPR() {
        ...
        // Create the builder object
        W3CEndpointReferenceBuilder builder = new
                           W3CEndpointReferenceBuilder();
```

```
        // Modify builder properties
        builder.address(otherServiceURI);

        // Create the endpoint reference from the builder object
        W3CEndpointReference epr = builder.build();
        return epr;
    }
...
```

The following code fragment uses the SubmissionEndpointReferenceBuilder class to obtain an endpoint reference that complies with the 2004/08 specification, and points to an endpoint that is in another module in this application:

```
...
@WebService(name="Calculator", targetNamespace="http://calculator.org")
public class Calculator {

    public W3CEndpointReference getEPR() {
        ...
        // Create the builder object
        SubmissionEndpointReferenceBuilder builder = new
                        SubmissionEndpointReferenceBuilder();

        // Modify builder properties
        builder.serviceName(calculatorService);
        builder.endpointName(calculatorPort);

        // Create the endpoint reference from the builder object
        SubmissionEndpointReference epr = builder.build();
        return epr;
    }
...
```

Results

You created an endpoint reference for use by your application.

What to do next

1. If required, convert the endpoint reference to an instance of the com.ibm.websphere.wsaddressing.EndpointReference class, by using the createIBMEndpointReference method. For example, on a client you might want to set the FaultTo message addressing property for outbound messages. You cannot set this property by using the JAX-WS API, so you must convert the endpoint reference representing the FaultTo endpoint to an instance of the com.ibm.websphere.wsaddressing.EndpointReference class, before setting it as a property on the BindingProvider object.
2. Continue with “Creating a JAX-WS web service application that uses Web Services Addressing” on page 1307.

Creating a JAX-RPC web service application that uses Web Services Addressing

Web Services Addressing (WS-Addressing) aids interoperability between web services by defining a standard way to address web services and provide addressing information in messages. This task describes the steps that are required to create a JAX-RPC web service that is accessed by using a WS-Addressing endpoint reference. The task also describes the extra steps that are required to use stateful resources as part of the web service.

Before you begin

The steps that are described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Complete this task if you are creating a web service that uses the WS-Addressing specification.

Procedure

1. Provide a web service interface, by creating or generating a Web Services Description Language (WSDL) document for the web service, that returns an endpoint reference to the target service. The interface must return an endpoint reference, which it can do by using a factory operation or a separate factory service. The target service can front a resource instance, for example a shopping cart.
2. Implement the web service created in the previous step. For the WS-Addressing portion of the implementation, complete the following steps:
   a. Create an endpoint reference that references the web service, by following the instructions in "Creating endpoint references by using the IBM proprietary Web Services Addressing API" on page 1316.
   b. Optional: If your interface involves a web service that fronts a resource instance, create or look up the resource instance.
   c. Optional: If you are using a resource instance, obtain the identifier of the resource and associate it with the endpoint reference as a reference parameter, by using the EndpointReference.setReferenceParameter(QName resource_id_name, String value) method. The resource identifier is application-dependent and might be generated during the creation of the resource instance.

      Attention: Do not put sensitive information in the resource identifier, because the identifier is propagated in the SOAP message.
      The endpoint reference now targets the resource.
   d. Return the endpoint reference.
3. If your web service uses resource instances, extend the implementation to match incoming messages to the appropriate resource instances. Because you associated the resource identifier with the endpoint reference that you created earlier, any incoming messages targeted at that endpoint reference contain the resource identifier information as a reference parameter in the SOAP header of the message. Because the resource identifier is passed in the SOAP header, you do not have to expose it on the web service interface. When WebSphere Application Server receives the message, it puts this information into the message context on the thread. Extend the implementation to undertake the following actions:
   a. Obtain the resource instance identifier from the message context, by using the EndpointReferenceManager.getReferenceParameterFromMessageContext(QName resource_id_name) method.
   b. Forward the message to the appropriate resource instance.
4. To configure a client to communicate with the service, use the endpoint reference that is produced by the service in the first step to send messages to the endpoint.
   a. Obtain a Stub object (by looking up the service in the Java Naming and Directory Interface (JNDI)), or create an empty Call object.
   b. Associate the endpoint reference with the proxy object. Use the setProperty(String property_name, Object value) method of the Stub or Call object. Use the WS-Addressing constant WSADDRESSING_DESTINATION_EPR as the property name, and the endpoint reference as the value.

      This procedure automatically configures the Stub or Call object, to represent the web service (or resource instance if your interface uses a web service that fronts a resource instance) of the endpoint reference. For Call objects, this process includes the configuration of the interface and endpoint metadata (portType and port elements) that are associated with the endpoint reference.

      Note: If the metadata of the endpoint reference conflicts with the information already associated with the outbound message, for example if the Stub object is configured to represent a different interface, a javax.xml.rpc.JAXRPCException exception is thrown on attempts to invoke the endpoint.

   Invocations on the Stub or Call object are now targeted at the web service or resource instance that is defined by the endpoint reference. When an invocation occurs, the product adds appropriate message

addressing properties, such as a reference parameter contained within the endpoint reference that identifies a target resource, to the message header.

Results

The web service and client are configured to use endpoint references through the WS-Addressing support.

Providing a web service interface that returns an endpoint reference to the target service

The following examples correspond to steps 1 to 4 in the procedure. The examples show how an IT organization might use web services to manage a network of printers. The organization might represent each printer as a resource that is addressed through an endpoint reference. The following examples show how to code such a service by using the IBM proprietary Web Services Addressing (WS-Addressing) application programming interfaces (APIs) that are provided by WebSphere Application Server, and JAX-WS.

The IT organization implements a PrinterFactory service that offers a CreatePrinter portType element. This portType element accepts a CreatePrinterRequest message to create a resource that represents a logical printer, and responds with an endpoint reference that is a reference to the resource.

The WSDL definition for such a PrinterFactory service might include the following code:

```
<wsdl:definitions targetNamespace="http://example.org/printer" ...
                  xmlns:pr=" http://example.org/printer">
  <wsdl:types>
    ...
    <xsd:schema...>
      <xsd:element name="CreatePrinterRequest"/>
      <xsd:element name="CreatePrinterResponse"
                   type="wsa:EndpointReferenceType"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="CreatePrinterRequest">
    <wsdl:part name="CreatePrinterRequest"
               element="pr:CreatePrinterRequest" />
  </wsdl:message>
  <wsdl:message name="CreatePrinterResponse">
    <wsdl:part name="CreatePrinterResponse"
               element="pr:CreatePrinterResponse" />
  </wsdl:message>
  <wsdl:portType name="CreatePrinter">
    <wsdl:operation name="createPrinter">
      <wsdl:input name="CreatePrinterRequest"
                  message="pr:CreatePrinterRequest" />
      <wsdl:output name="CreatePrinterResponse"
                   message="pr:CreatePrinterResponse" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

The CreatePrinter operation in the previous example returns a wsa:EndpointReference object that represents the newly created Printer resource. The client can use this endpoint reference to send messages to the service instance that represents the printer.

The createPrinter method shown in the following example creates an endpoint reference to the Printer service. The operation then obtains the identifier for the individual printer resource instance, and associates it with the endpoint reference. Finally, the createPrinter method converts the EndpointReference object, which now represents the new printer, into a W3CEndpointReference object, and returns the converted endpoint reference.

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
import com.ibm.websphere.wsaddressing.EndpointReference;
import com.ibm.websphere.wsaddressing.jaxws.EndpointReferenceConverter;
import com.ibm.websphere.wsaddressing.jaxws.W3CEndpointReference;

import javax.xml.namespace.QName;

public class MyClass {

    // Create the printer
    ...
```

```
    // Define the printer resource ID as a static constant as it is required in later steps
    public static final QName PRINTER_ID_PARAM_QNAME = new QName("example.printersample",
                                                        "IBM_WSRF_PRINTERID", "ws-rf-pr" );
    public static final QName PRINTER_SERVICE_QNAME = new QName("example.printer.com", "printer", "...");
    public static final String PRINTER_ENDPOINT_NAME = new String("PrinterService");

    public W3CEndpointReference createPrinter(java.lang.Object createPrinterRequest)
    throws Exception {
     // Create an EndpointReference that targets the appropriate WebService URI and port name.
     EndpointReference epr =  EndpointReferenceManager.createEndpointReference(PRINTER_SERVICE_QNAME,
                                                                    PRINTER_ENDPOINT_NAME);

     // Create or lookup the stateful resource and derive a resource
     // identifier string.
     String resource_identifier = "...";

     // Associate this resource identifier with the EndpointReference as
     // a reference parameter.
     // The choice of name is arbitrary, but should be unique
     // to the service.
     epr.setReferenceParameter(PRINTER_ID_PARAM_QNAME,resource_identifier);
     // The endpoint reference now targets the resource rather than the service.
     ...

     return EndpointReferenceConverter.createW3CEndpointReference(epr);
    }
}
```

Because of the web service implementation described previously, the printer resource instance now has a unique identifier embedded in its endpoint reference. This identifier becomes a reference parameter in the SOAP header of subsequent messages that are targeted at the web service, and can be used by the web service to match incoming messages to the appropriate printer.

When a web service receives a message containing WS-Addressing message-addressing properties, the WebSphere Application Server processes these properties before the message is dispatched to the application endpoint, and sets them into the message context on the thread. The Printer web service application accesses the reference parameters that are associated with the target endpoint from the WebServiceContext object, as illustrated in the following example:

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
...
 // Initialize the reference parameter name
 QName name = new QName(..);
 // Extract the String value.
 String resource_identifier =
        EndpointReferenceManager.getReferenceParameterFromMessageContext(PRINTER_ID_PARAM_QNAME);
```

The web service implementation can forward messages based on the printer identity acquired from the getReferenceParameterFromMessageContext method to the appropriate printer instances.

The client creates a JAX-WS proxy for the printer, and converts the proxy into a BindingProvider object. The client then associates the EndpointReference object obtained previously with the request context of the BindingProvider object, as illustrated in the following example.

```
import javax.xml.ws.BindingProvider;
...

 javax.xml.ws.Service service= ...;
 Printer myPrinterProxy = service.getPort(portName, Printer.class);

 javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider)myPrinterProxy;

 // Retrieve the request context for the BindingProvider object
 Map myMap = myBindingProvider.getRequestContext();

 // Associate the endpoint reference that represents the new printer to the request context
 // so that the BindingProvider object now represents a specific printer instance.
 myMap.put(WSADDRESSING_DESTINATION_EPR, destinationEpr);

...
```

The BindingProvider object now represents the new printer resource instance, and can be used by the client to send messages to the printer through the Printer web service. When the client invokes the

BindingProvider object, WebSphere Application Server adds appropriate message-addressing properties to the message header, which in this case is a reference parameter contained within the endpoint reference that identifies the target printer resource.

Alternatively, the client can use a JAX-RPC Stub or Call object, which the client configures to represent the new printer. The use of the Call object is illustrated in the following example.

```
import javax.xml.rpc.Call;
...
 :
 // Associate the endpoint reference that represents the new printer with the call.
 call.setProperty(
        "com.ibm.websphere.wsaddressing.WSAConstants.
                              WSADDRESSING_DESTINATION_EPR ", epr);
```

From the perspective of the client, the endpoint reference is opaque. The client cannot interpret the contents of any endpoint reference parameters and should not try to use them in any way. Clients cannot directly create instances of endpoint references because the reference parameters are private to the service provider; clients must obtain endpoint references from the service provider, for example through a provider factory service, and then use them to direct web service operations to the endpoint that is represented by the endpoint reference, as shown.

What to do next

- Refer to “Web Services Addressing security” on page 1361 for information about security with WS-Addressing.
- Deploy the application. For this scenario, you do not have to take any additional steps to enable the WS-Addressing support in WebSphere Application Server because you specified a WS-Addressing property on the client. For more information, and for other scenarios that might require additional steps, see “Enabling Web Services Addressing support for JAX-RPC applications” on page 1365.

Creating endpoint references by using the IBM proprietary Web Services Addressing API

Endpoint references are a primary concept of the Web Services Addressing (WS-Addressing) interoperability protocol, and provide a standard mechanism to encapsulate information about specific Web service endpoints. This product provides interfaces for you to create endpoint references by using the IBM proprietary implementation of the WS-Addressing standard.

About this task

This task is a subtask of “Creating a JAX-RPC web service application that uses Web Services Addressing” on page 1312.

Complete this task if you are writing an application that uses the IBM proprietary WS-Addressing API. Such applications require endpoint references to target web service endpoints. When you are writing the application, you might not know the address of the endpoint, because the address can change when the application is deployed. By using the IBM proprietary API, you can either specify the endpoint address, or allow the product to generate it for you at run time.

You can also specify the behavior of endpoint references in a cluster environment.

If you want to use endpoint reference objects from the standard JAX-WS API instead of the IBM proprietary equivalents, but want the extra functions provided by the IBM proprietary API, create the endpoint references by using the methods described in this task and then convert them by using the supplied converter classes. For example, you might want to undertake such a conversion if you have a JAX-WS service application and you are creating endpoint references that represent stateful session beans, or that have an affinity to a particular server, or are workload managed. You cannot create such endpoint references by using the JAX-WS API.

Procedure

- To create an endpoint reference with an address that you specify directly, use the WS-Addressing EndpointReferenceManager.createEndpointReference(URI address) method of the system programming interface (SPI) provided. This method is useful in test scenarios, where the address of the service does not change.
- To create an endpoint reference with an address that is automatically generated by the product, complete the following steps:
  1. If you created the web service deployment descriptor file, webservices.xml, manually, ensure that the webservice-description-name in the file is the same as the local part of the Web Services Description Language (WSDL) service name. If you generated the webservices.xml file by using the tools provided, the names match by default. This match is required for the generation of the correct URI for the endpoint reference.
  2. Create the endpoint reference by using the method that is appropriate for the object that the reference will represent.
     - If you are creating an endpoint reference to represent a stateful session bean that maintains in-memory state, create the endpoint reference using the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName, Remote statefulSessionBean) method of the application programming interface (API) provided. This method ensures that requests are targeted at the specific server that hosts the stateful session bean instance, and are not workload-managed.

       Also, if high availability for stateful session beans is specified, the endpoint reference remains valid even if the stateful session bean is failed over.

       Note: Affinity to stateful session beans that use this method is not supported on the z/OS operating system. If a highly available stateful session bean fails over to another control region, or is passivated from one servant region and reactivated on another servant region in the same control region, the endpoint reference is no longer valid. To achieve affinity with a stateful session bean, run the application in a transaction to use transactional context affinity, or use HTTP session affinity.
     - If you are creating an endpoint reference to represent any other object, create the endpoint reference by using the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method of the API. The combination of service name and endpoint name must be unique in the server. If there is more than one web service application with the same service name and endpoint name, the application server cannot generate a unique URI object for the endpoint. If you cannot ensure that the combination of service name and endpoint name is unique, use an SPI method to create the endpoint reference.

       Because the endpoint reference might be workload-managed at a later date, ensure that the endpoint does not contain any in-memory state. For JAX-WS applications, you can prevent the endpoint reference from being workload-managed by attaching a WS-Addressing policy set to the application and configuring the policy set binding to prevent the workload management of referenced endpoints in clusters.

       Note: You can use the EndpointReferenceManager.createEndpointReference(QName serviceName, String endpointName) method to create an endpoint reference for any endpoint reference in the application server. This behavior differs from that of the JAX-WS API, where creating an endpoint reference by using the service name and endpoint name is restricted to endpoints within the same Java EE application.

     When the application invokes either of the previous two methods, the product generates the address URI for the endpoint reference, and puts the service name and endpoint name into the metadata of the newly created endpoint reference.

     Note: If you configured a virtual host for the server on which the endpoint is created, the URI of the endpoint reference refers to the virtual host of the HTTP server configuration. You can use

the administrative console to override this setting and provide your own HTTP endpoint URL information. The methods described previously will use the overridden value to generate the address URI for the endpoint reference.

Results

You created an endpoint reference for use by your application.

What to do next

1. If you want to convert the endpoint references from IBM proprietary WS-Addressing objects to standard JAX-WS WS-Addressing objects, use one of the following methods of the com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceConverter class, depending on the namespace of the endpoint reference:
   - createW3CEndpointReference(EndpointReference epr): use this method if the EndpointReference object uses the 2005/08 specification. This method creates a W3CEndpointReference object.
   - createSubmissionEndpointReference(EndpointReference epr): use this method if the EndpointReference object uses the 2004/08 specification. This method creates a SubmissionEndpointReference object.
2. Continue with “Creating a JAX-RPC web service application that uses Web Services Addressing” on page 1312, or if you converted the endpoint reference to the standard JAX-WS API, continue with “Creating a JAX-WS web service application that uses Web Services Addressing” on page 1307.

Example: Creating a web service that uses the JAX-WS Web Services Addressing API to access a generic web service resource instance

Consider an IT organization that has a network of printers that it wants to manage using web services. The organization might represent each printer as a resource that is addressed through an endpoint reference. This example shows how to code such a service by using the JAX-WS Web Services Addressing (WS-Addressing) application programming interfaces (APIs) that are provided by WebSphere Application Server.

Providing a web service interface that returns an endpoint reference to the target service

The IT organization implements a PrinterFactory service that offers a CreatePrinter portType element. This portType element accepts a CreatePrinterRequest message to create a resource that represents a logical printer, and responds with an endpoint reference that is a reference to the resource.

The WSDL definition for such a PrinterFactory service might include the following code:

```
<wsdl:definitions targetNamespace="http://example.org/printer" ...
                  xmlns:pr=" http://example.org/printer">
  <wsdl:types>
    ...
    <xsd:schema...>
      <xsd:element name="CreatePrinterRequest"/>
      <xsd:element name="CreatePrinterResponse"
                   type="wsa:EndpointReferenceType"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="CreatePrinterRequest">
    <wsdl:part name="CreatePrinterRequest"
               element="pr:CreatePrinterRequest" />
  </wsdl:message>
  <wsdl:message name="CreatePrinterResponse">
    <wsdl:part name="CreatePrinterResponse"
               element="pr:CreatePrinterResponse" />
  </wsdl:message>
  <wsdl:portType name="CreatePrinter">
    <wsdl:operation name="createPrinter">
      <wsdl:input name="CreatePrinterRequest"
                  message="pr:CreatePrinterRequest" />
      <wsdl:output name="CreatePrinterResponse"
```

```
                message="pr:CreatePrinterResponse" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

The CreatePrinter operation in the previous example returns a wsa:EndpointReference object that represents the newly created Printer resource. The client can use this endpoint reference to send messages to the service instance that represents the printer.

Implementing the web service interface

The createPrinter method shown in the following example obtains the identifier for the individual printer resource instance. The operation then creates an endpoint reference to the Printer service, and associates the printer ID with the endpoint reference. Finally, the createPrinter method returns the endpoint reference.

```
import javax.xml.ws.wsaddressing.W3CEndpointReference;
import javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder;

import javax.xml.namespace.QName;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class MyClass {

  // Create the printer
  ...

  //Define the printer resource ID as a static constant as it is required in later steps
    public static final QName PRINTER_SERVICE_QNAME = new QName("example.printer.com", "printer", "...");
    public static final QName PRINTER_ENDPOINT_NAME = new QName("example.printer.com", "PrinterService", "...");

    public W3CEndpointReference createPrinter(java.lang.Object createPrinterRequest)
    {
     Document document = ...;

     // Create or lookup the stateful resource and derive a resource
     // identifier string.
     String resource_identifier = "...";

     // Associate this resource identifier with the EndpointReference as
     // a reference parameter.
     // The choice of name is arbitrary, but should be unique
     // to the service.
     Element element = document.createElementNS("example.printersample",
             "IBM_WSRF_PRINTERID");
     element.appendChild( document.createTextNode(resource_identifier) );
     ...

     // Create an EndpointReference that targets the appropriate WebService URI and port name.
     // Alternatively, the getEndpointReference() method of the MessageContext can be used.
     W3CEndpointReferenceBuilder builder = new W3CEndpointReferenceBuilder();
     builder.serviceName(PRINTER_SERVICE_QNAME);
     builder.endpointName(PRINTER_ENDPOINT_NAME);
     builder.referenceParameter(element);

     // The endpoint reference now targets the resource rather than the service.
     return builder.build();
    }
}
```

Extending the target service to match incoming messages to web service resource instances

Because of the web service implementation described previously, the printer resource instance now has a unique identifier embedded in its endpoint reference. This identifier becomes a reference parameter in the SOAP header of subsequent messages that are targeted at the web service, and can be used by the web service to match incoming messages to the appropriate printer.

When a web service receives a message containing WS-Addressing message addressing properties, the WebSphere Application Server processes these properties before the message is dispatched to the application endpoint, and sets them into the message context on the thread. The Printer web service application accesses the reference parameters that are associated with the target endpoint from the WebServiceContext object, as illustrated in the following example:

```
@Resource
private WebServiceContext context;
...
List list = (List) context.getMessageContext().get(MessageContext.REFERENCE_PARAMETERS);
```

If your application uses the 2004/08 version of the WS-Addressing specification, use the IBM proprietary API to retrieve the message parameters, as illustrated in the following example.

```
import com.ibm.websphere.wsaddressing.EndpointReferenceManager;
...
 // Initialize the reference parameter name
 QName name = new QName(..);
 // Extract the String value.
 String resource_identifier =
        EndpointReferenceManager.getReferenceParameterFromMessageContext(PRINTER_ID_PARAM_QNAME);
```

The web service implementation can forward messages based on the printer ID to the appropriate printer instances.

Using endpoint references to send messages to an endpoint

The client uses the endpoint reference returned from the service to create a JAX-WS proxy for the printer, as illustrated in the following example.

```
javax.xml.ws.Service jaxwsServiceObject= ...;
W3CEndpointReference epr = ...;
...
Printer myPrinterProxy = jaxwsServiceObject.getPort(epr, Printer.class, new AddressingFeature());
```

The proxy object now represents the new printer resource instance, and can be used by the client to send messages to the printer through the Printer Web service. When the client invokes the service, WebSphere Application Server adds appropriate message addressing properties to the message header, which in this case is a reference parameter contained within the endpoint reference that identifies the target printer resource.

From the perspective of the client, the endpoint reference is opaque. The client cannot interpret the contents of any endpoint reference parameters and should not try to use them in any way. Clients cannot directly create instances of endpoint references because the reference parameters are private to the service provider; clients must obtain endpoint references from the service provider, for example through a provider factory service, and then use them to direct web service operations to the endpoint that is represented by the endpoint reference, as shown.

Using the IBM proprietary Web Services Addressing SPIs: Performing more advanced Web Services Addressing tasks

This product provides proprietary system programming interfaces for more advanced Web Services Addressing (WS-Addressing) tasks, which involve the WS-Addressing message-addressing properties that are passed in the SOAP header of a web service message. You can also use the SPIs to choose a WS-Addressing specification level other than the default used by the product.

Before you begin

You cannot use the standard JAX-WS API classes with these proprietary SPIs. However, you can convert endpoint references created using the standard JAX-WS API classes to instances of the com.ibm.websphere.wsaddressing.EndpointReference class, by using the com.ibm.websphere.wsaddressing.jaxws21.EndpointReferenceConverter class. You can then use these converted endpoint references with the SPIs.

The steps described in this task apply to servers and clients that run on WebSphere Application Server.

About this task

Complete this task to specify or acquire WS-Addressing message-addressing properties, or if you have an application that needs to interoperate with a client or endpoint that is not using the default WS-Addressing specification supported by this product.

Procedure

- To manipulate message-addressing properties, follow the instructions in "Specifying and acquiring message-addressing properties by using the IBM proprietary Web Services Addressing SPIs"
- To interoperate with the pre-W3C specification of WS-Addressing, with the namespace http://schemas.xmlsoap.org/ws/2004/08/addressing, refer to "Interoperating with Web Services Addressing endpoints that do not support the default specification supported by WebSphere Application Server" on page 1322.

Specifying and acquiring message-addressing properties by using the IBM proprietary Web Services Addressing SPIs

Using the proprietary Web Services Addressing (WS-Addressing) system programming interfaces (SPIs), you can add WS-Addressing message addressing properties (MAPs) to the SOAP headers of an outbound client message, through properties on the JAX-WS BindingProvider request context, or the JAX-RPC Stub or Call object. When the target endpoint receives the message, the SPI enables the endpoint to acquire the MAPs through properties on the message context.

About this task

There are no equivalent SPIs in the JAX-WS standard. If you want to set message-addressing properties in a client that uses JAX-WS endpoint references, you must convert the endpoint references to the IBM proprietary classes, before using them with these SPIs.

Complete this task if you are a web service developer that uses the WS-Addressing support, or a system programmer that uses the IBM proprietary WS-Addressing SPIs to specify message addressing properties, such as fault or reply endpoint references, on web services messages.

The properties that you can set or retrieve are described, with the Java type of property instances, in IBM proprietary Web Services Addressing SPIs. Most properties are of type com.ibm.websphere.wsaddressing.EndpointReference, for example destination, reply, or fault endpoint references. The relationship property is a java.util.Set object that contains instances of the com.ibm.wsspi.wsaddressing.Relationship class. Use relationships when you want to specify an association between messages; for example, in a response message you might want to specify the ID of the message to which you are replying. The action property is an AttributedURI object that identifies a specific method or operation within the target endpoint.

Attention: The destination endpoint reference and action properties are required for the message to be WS-Addressing compliant.

Procedure

1. On the client, obtain the endpoint reference from the service and associate it with your BindingProvider object request context, or your Stub or Call object, as described in "Creating a JAX-RPC web service application that uses Web Services Addressing" on page 1312.
2. Create instances of the required properties. For example, if you want to specify an endpoint reference for the target service to send replies to, create an instance of the com.ibm.websphere.wsaddressing.EndpointReference class, to use as the WSADDRESSING_REPLYTO_EPR property.

3. Set the required properties by associating them with the BindingProvider object request context, or the Stub or Call object. If you are using a Stub or Call object, use the setProperty(String property_name, Object value) method. Note that unlike the endpoint reference required for the first step, these endpoint references do not have to be converted to another type, because they are passed in the header of the SOAP message rather than the body. The following example sets a destination endpoint reference and a reply endpoint reference on a BindingProvider object request context:

```
import javax.xml.ws.BindingProvider;
...
 javax.xml.ws.Service jaxwsServiceObject=...;
 Printer myPrinterProxy = jaxwsServiceObject.getPort(portName, Printer.class);

 javax.xml.ws.BindingProvider myBindingProvider = (javax.xml.ws.BindingProvider)myPrinterProxy;

 // Retrieve the request context for the BindingProvider object
 Map myMap = myBindingProvider.getRequestContext();

 // Associate the endpoint reference for the web service. This property is required for the message
 // to be WS-Addressing compliant.
  myMap.put(WSADDRESSING_DESTINATION_EPR, destinationEpr);

 // Associate the endpoint reference that represents the reply to the request context
 myMap.put(WSADDRESSING_REPLYTO_EPR, replyToEpr);
```

   When an invocation occurs on the BindingProvider, Stub, or Call object, the product adds the appropriate MAPs to the message header.

4. On the server, retrieve the MAPs from the inbound message through the javax.xml.ws.WebServiceContext or javax.xml.rpc.handler.MessageContext object that is currently on the thread. When WebSphere Application Server receives the message, it puts the MAP information into the message context on the thread, making it available to the service. You can retrieve the message context by, for example, using the session context of the endpoint enterprise bean. For more information about message contexts, refer to the JSR-109 standard. The following example retrieves the reply endpoint reference by using the web service context:

```
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.WebServiceContext;
...

  // Obtain the message context from the WebService context
  private WebServiceContext wsContext;
  MessageContext context = wsContext.getMessageContext();

  // Retrieve the reply endpoint reference
  replyToEpr = context.getProperty(WSADDRESSING_INBOUND_REPLYTO_EPR);
```

Interoperating with Web Services Addressing endpoints that do not support the default specification supported by WebSphere Application Server

A target web service endpoint might not support the same Web Services Addressing (WS-Addressing) namespace as this product. In most cases, you do not have to undertake any extra actions to interoperate with such endpoints, however some scenarios require additional steps in the implementation of your web service.

About this task

WebSphere Application Server supports the default WS-Addressing 2005/08 namespace http://www.w3.org/2005/08/addressing. Complete this task to interoperate with endpoints that support other namespaces. This task specifically describes interoperation with endpoints that are hosted on a node that supports only the 2004/08 namespace: http://schemas.xmlsoap.org/ws/2004/08/addressing.

If you are using the standard JAX-WS API, ensure that you use the appropriate feature, annotation or endpoint reference class for the 2004/08 namespace.

If you are sending to or receiving messages from an endpoint that supports only the 2004/08 namespace, you do not have to undertake any additional steps for interoperability. This product recognizes and understands incoming WS-Addressing messages that conform to the 2004/08 specification, and outbound

messages automatically adhere to the namespace of their destination endpoint reference. If you are sending a request, all WS-Addressing elements, such as reply endpoint or fault endpoint elements, must use the same namespace as the message. Any discrepancy results in a JAX-WS or JAX-RPC configuration error.

If you are interacting in a different way with an endpoint that supports only the 2004/08 namespace, such as exporting endpoint references in the message header or body, and you are not using the JAX-WS standard API, you must undertake additional steps as detailed below.

Procedure

- If you are generating a web service for use by a client that supports only the 2004/08 specification, update the WS-Addressing namespace in the Web Services Description Language (WSDL) document for your web service, by changing `http://www.w3.org/2006/05/addressing/wsdl` to `http://schemas.xmlsoap.org/ws/2004/08/addressing`.

  Note: Only the WS-Addressing WSDL Action extensibility element is recognized by pre-W3C WS-Addressing clients.

- If you are creating endpoint references at run time for export to an endpoint that supports the 2004/08 namespace only, perform the following steps:
  1. Create the endpoint reference to export.
  2. Associate the appropriate namespace with the endpoint reference, by using the setNamespace method. The following example illustrates the association of the 2004/08 namespace with an endpoint reference:

```
import com.ibm.wsspi.wsaddressing.EndpointReference;
import com.ibm.wsspi.wsaddressing.NamespaceNotSupportedException;
import com.ibm.wsspi.wsaddressing.WSAConstants;

:

EndpointReference epr = ...

try
{
    epr.setNamespace(WSAConstants.WSADDRESSING_NAMESPACE_2004_08);
} catch (NamespaceNotSupportedException e)
{
    // Error handling code here
}
```

  When you pass the endpoint reference to the target endpoint, in either the SOAP body or the SOAP header of a message, the endpoint reference is appropriately serialized into SOAP elements according to its namespace.

- To establish the namespace of an inbound request, use the IBM proprietary WS-Addressing system programming interface (SPI) to retrieve the WSADDRESSING_INBOUND_NAMESPACE property from the inbound message context. This property specifies the Core WS-Addressing specification namespace of the incoming message.

  Note: This procedure uses the IBM proprietary WS-Addressing API. There is no equivalent procedure in the JAX-WS API.

  You can retrieve the message context by, for example, using the session context of the endpoint enterprise bean. For more information about message contexts, refer to the JSR-109 specification. The following code example shows how you can establish the namespace of an incoming message on the receiving endpoint:

```
import com.ibm.wsspi.wsaddressing.WSAConstants;
import javax.xml.rpc.handler.MessageContext;

:
  // If the endpoint is implemented as an enterprise bean, you can use its session context
  // to obtain the message context
  private SessionContext sessionContext;
  MessageContext context = sessionContext.getMessageContext();

  try
  {
```

```
    String namespace = (String)msgContext.getProperty(WSAConstants.WSADDRESSING_INBOUND_NAMESPACE);
  } catch (IllegalArgumentException e)
  {
    // Error handling code here
  }
```

Enabling Web Services Addressing support for JAX-WS applications

The Web Services Addressing (WS-Addressing) support provides mechanisms to address web services and provide addressing information in messages. For JAX-WS applications, you can enable WS-Addressing support in several different ways, such as configuring policy sets or using annotations in code.

About this task

Perform this task to enable the WS-Addressing support, either as a service provider or as a client of a service provided by another party.

For service providers, WS-Addressing support is enabled by default, so you do not have to undertake any actions to enable support. However, you can use the enabling mechanisms to modify other WS-Addressing behavior for the service, such as whether WS-Addressing information is required, and what is included in the generated WSDL document.

- Modify the behavior of the WS-Addressing support after the application is deployed by attaching a policy set to the application. Within the policy set, you can configure the WS-Addressing policy type to specify whether WS-Addressing information is required in incoming messages, and whether to use synchronous or asynchronous messaging. You can communicate the WS-Addressing policy configuration to other servers and clients that support WS-Policy, by enabling policy sharing on the server, and by applying the provider policy on the client. This method overrides other methods.
- Use deployment descriptor elements within a `port-component` element.
- Modify the behavior of the WS-Addressing support during development of the service by using the Addressing or SubmissionAddressing annotations in the service code. Within each annotation you can specify whether WS-Addressing is enabled on the server, whether the server requires WS-Addressing information in incoming messages and the message exchange pattern the server will use. The presence of the Addressing annotation in the code adds a UsingAddressing element and a WS-Policy assertion to any WSDL document that is generated for the service.
- Modify the behavior of the WS-Addressing support during development of the service by either adding UsingAddressing elements or WS-Policy assertions into the WSDL document. If you do provide your own WSDL document, instead of relying on the JAX-WS runtime environment to generate one, and your WSDL document is being consumed by non JAX-WS 2.2 clients, you may wish to include the UsingAddressing element as such clients will not process WS-Policy assertions.

For service clients, WS-Addressing support is disabled by default. Use one of the following methods to enable WS-Addressing support. The following list is in descending order of precedence, because you can apply multiple methods. For example the deployment descriptor method overrides the use of annotations or features, but is itself overridden by the use of policy sets.

- Attach a policy set to the client artifact and perform one of the following actions:
  - Specify in the policy that WS-Addressing is mandatory.
  - Apply both client and provider policies. In this situation, WS-Addressing support is enabled on the client only if policy sharing is enabled on the service provider and the policy configuration for the provider specifies that WS-Addressing is mandatory.

  This method overrides other methods.
- Set the **`com.ibm.websphere.webservices.use.async.mep`** property on the client request context.
- Use the IBM proprietary WS-Addressing SPI to add message-addressing properties to the message request context.
- Use deployment descriptor elements within a `port-component-ref` element.

- Use the Addressing annotation in combination with the WebServiceRef annotation in the client code (when you are using an injected Web services proxy reference).
- Use addressing features in the client code. Properties set by using this method override those set in the WSDL document for the service.
- Use WS-Policy assertions in the WSDL document to specify WS-Addressing support. If you add Addressing annotations to your client application code, the generated WSDL document will contain WS-Policy assertions.
- Specify the UsingAddressing element in the WSDL document for the service. If the service uses the Addressing annotation and you generate the WSDL document from the code, the UsingAddressing element already exists.

The following tables summarize the behavior of the WS-Addressing support. Use this table to determine whether a request message is accepted for client settings that do not involve policy configuration.

This is a complex table containing spanned column headings. There is a header column on the left entitled "Client settings" which specifies whether WS-Addressing support is enabled on the client and also which messaging style is configured. The second main column shows the policy settings on the provider, with two sub-columns for whether WS-Addressing is optional or mandatory. Both of these sub-columns are further sub-divided into three sub-columns for messaging style (synchronous, asynchronous, or both). The third main column shows the WSDL settings on the provider, with two sub-columns for whether WS-Addressing is optional or mandatory.

Table 138. How client and provider settings interact to determine whether a request message is accepted. The preceding paragraph describes this table.

	Provider policy settings						**Provider WSDL settings (UsingAddressing `required` attribute)**	
	WS-Addressing is optional			**WS-Addressing is mandatory[1]**				
Client settings	**Synchronous and asynchronous**	**Synchronous only**	**Asynchronous only**	**Synchronous and asynchronous**	**Synchronous only**	**Asynchronous only**	**`false` (WS-Addressing is optional)**	**`true` (WS-Addressing is mandatory[1])**
WS-Addressing support enabled and messaging style synchronous	Message accepted	Message accepted	Fault	Message accepted	Message accepted	Fault	Message accepted	Message accepted
WS-Addressing support enabled and messaging style asynchronous	Message accepted	Fault	Message accepted	Message accepted	Fault	Message accepted	Message accepted	Message accepted
WS-Addressing support not enabled and messaging style synchronous	Message accepted	Message accepted	Message accepted[2]	Fault	Fault	Fault	Message accepted	Fault
WS-Addressing support not enabled and messaging style asynchronous[3]	Fault	Fault	Fault	Fault	Fault	Fault	Fault	Fault

Notes:

1. If WS-Addressing is mandatory, all requests without WS-Addressing headers are rejected.

2. The messaging style is only enforced if WS-Addressing headers are present in the request.
3. Asynchronous messaging is not possible without WS-Addressing headers.

Use the following table to determine whether a request message is accepted when the client and provider both have a WS-Addressing policy configuration; the client has provider and client policies applied; and policy sharing is enabled on the server.

This is a complex table containing spanned column headings. There is a header column on the left entitled "Client settings" which specifies whether WS-Addressing support is optional or mandatory on the client and also which messaging style is configured. The second main column shows the policy settings on the provider, with two sub-columns for whether WS-Addressing is optional or mandatory. Both of these sub-columns are further sub-divided into three sub-columns for messaging style (synchronous and asynchronous, synchronous only, or asynchronous only).

Table 139. How client and provider policy settings interact to determine whether a request message is accepted. The preceding paragraph describes this table.

	Provider policy settings					
	WS-Addressing is optional			**WS-Addressing is mandatory**		
Client policy settings	**Synchronous and asynchronous**	**Synchronous only**	**Asynchronous only**	**Synchronous and asynchronous**	**Synchronous only**	**Asynchronous only**
WS-Addressing optional and messaging style synchronous and asynchronous	Message accepted	Message accepted	Message accepted	Message accepted	Message accepted	Message accepted
WS-Addressing optional and messaging style synchronous only	Message accepted	Message accepted	Message accepted[1]	Message accepted	Message accepted	Fault
WS-Addressing optional and messaging style asynchronous only[2]	Fault	Fault	Fault	Message accepted	Fault	Message accepted
WS-Addressing mandatory and messaging style synchronous and asynchronous	Message accepted	Message accepted	Message accepted	Message accepted	Message accepted	Message accepted
WS-Addressing mandatory and messaging style synchronous only	Message accepted	Message accepted	Fault	Message accepted	Message accepted	Fault
WS-Addressing mandatory and messaging style asynchronous only	Message accepted	Fault	Message accepted	Message accepted	Fault	Message accepted

Notes:

1. The messaging style is only enforced if WS-Addressing headers are present in the request.
2. Asynchronous messaging is not possible without WS-Addressing headers.

If the provider and client policies are not shared, the client does not send WS-Addressing headers (unless you enable WS-Addressing on the client by another method). In this situation, if the provider policy specifies that WS-Addressing is mandatory, the server generates a fault regardless of the messaging style.

Procedure

- To modify the behavior of the WS-Addressing support by creating or modifying policy sets on either the service provider or client, see the topic: “Enabling Web Services Addressing support for JAX-WS applications using policy sets” on page 1327.

- Set the **com.ibm.websphere.webservices.use.async.mep** property on the client request context when using WebSphere Application Server clients to enable WS-Addressing support. See the topic: “Invoking JAX-WS web services asynchronously” on page 1216 for more information.
- To modify the behavior of the WS-Addressing support by using the deployment descriptor of the service or client application, see the topic: “Enabling Web Services Addressing support for JAX-WS applications using deployment descriptors” on page 1354.
- To modify the behavior of the WS-Addressing support programmatically by using addressing annotations in the service application, or on the client with an injected web service proxy reference, see the topic: “Enabling Web Services Addressing support for JAX-WS applications using addressing annotations” on page 1355.
- To enable WS-Addressing support programmatically on the client by creating an instance of an addressing feature class, see the topic: “Enabling Web Services Addressing support for JAX-WS applications using addressing features” on page 1357.
- To modify the behavior of the WS-Addressing support during the development of a client or service application by adding WS-Policy assertions into the WSDL file, see the topic: “Enabling Web Services Addressing support for JAX-WS applications using WS-Policy” on page 1358.

Results

WS-Addressing properties are now included in the SOAP message header, and are processed by the server on receipt of the message.

Enabling Web Services Addressing support for JAX-WS applications using policy sets

For JAX-WS applications, you can enable WS-Addressing support after you deploy an application to the server, by creating or modifying policy sets, and attaching those policy sets to either a service or client application.

About this task

You can also configure the WS-Addressing support using other methods, for example in the code of the application, however this method overrides all other methods. Note that WS-Addressing support is enabled by default for service providers.

Procedure

1. Ensure that you have a policy set that contains the WS-Addressing policy type. If you have to create a new policy set, or add the WS-Addressing policy to an existing policy set, refer to Managing policy sets using the administrative console for instructions.
2. Configure the WS-Addressing policy type according to the instructions in “Configuring the WS-Addressing policy” on page 1335. Use the two settings to specify whether WS-Addressing is mandatory, and whether to use a synchronous or asynchronous message exchange pattern. The default settings are that WS-Addressing is not mandatory, and that both synchronous and asynchronous messaging patterns are used.
3. Attach the policy set to a web service provider or client artifact, according to the instructions in “Attaching a policy set to a service artifact” on page 1337.
4. Optional: If you want to communicate the WS-Addressing policy settings to other servers and clients, configure policy sharing as described in “Configuring a service provider to share its policy configuration” on page 1338 or “Configuring the client policy to use a service provider policy” on page 1346. If policy sharing is enabled and the server and client cannot agree a policy, normal WS-Policy behavior applies (a policy error is produced).

Results

WS-Addressing properties are now included in the SOAP message header, and are processed by the server on receipt of the message.

Creating policy sets using the administrative console

You can use the administrative console to either create a policy set by specifying all the necessary information or by copying an existing policy set that you rename. You can use policy sets, or assertions that define services, to simplify your web services configuration because policy sets group security and other web services settings into reusable units.

Before you begin

To create a new policy set, you can either specify the information to create a new policy set or you can copy and rename an existing policy set. Using either method, you need basic information about the policy set that you want to create, such as the name, description, policies to include, policy details, attachments, and binding configurations. If you are creating a policy set by copying an existing policy set, then you should also view the existing policy sets to choose one with properties that are most similar to the one you plan to create.

About this task

Whether you choose to create a new policy set or copy and rename an existing one, start from the Applications policy sets collection in the administrative console.

Procedure

1. From the administrative console, click **Services > Policy sets > Application policy sets** or **Services > Policy sets > System policy sets**.
2. If the policy set you are creating is:
   - a new policy set, then click **New**.
   - an existing policy set to be copied and renamed, click the **Select** box beside the name of the policy set to be copied in the **Name** column and click **Copy**.

   Using either method, this action opens the Policy set settings view to specify the required information about the policy set being created or copied.
3. Enter the name of the policy set that you want to create or copy in the **Name** field.
4. Enter a brief description of the policy set in the **Description** field. This is the description that displays in the Application policy sets or the System policy set collection, so it must be meaningful to you and other potential users of this policy set.

   Note: If you created a new policy set, it does not contain policies to edit until you add them to the policy set. The policy set is initially empty.

Results

You have provided the basic information to create a policy set.

Example

After you have looked at your web services, you might decide that the WS-I RSP default policy set most closely meets your needs. You would go to the administrative console and click **Services > Policy sets > Application policy sets** to access the Application policy sets collection. Locate the WS-I RSP default in the **Name** column of the table and click the box beside it (in the **Select** column). Click **Copy**. This opens the Policy set settings window. You might want to name your policy set by your company or division so you could provide a name like `ABC WS-I RSP` in the Name field. Because you know others in your

organization might access and use it, you've chosen a name that is meaningful to those people too. You want to be sure everyone knows exactly what this copy of the WS-I RSP policy is used for, so you add a description in the **Description** field describing it. Now you want to customize the policy set so you edit the policy information by clicking the name of a policy to edit it.

When you identify the requirements of your web service, you might decide that none of the default policy sets meet your needs closely enough to use them as a template so you might decide to create your own policy set. You would first create the policy set with the name you choose to give it. As if you were reusing an existing template, you would go to the administrative console and click **Services > Policy sets > Application policy sets** and click **New**. The Policy set settings window opens but note that the Policy set name field is blank and there are not yet any associated policies in the table. Enter the name and add any policies necessary.

When you add policies to a policy set, the policies are set to their default values. You can then edit the policies to modify any attribute values that need to be changed and save the settings.

What to do next

If you are creating a new policy set without copying an existing policy set, you need to specify the policy information. If you are copying an existing policy set, you can either accept the default policies associated with the policy set or you can change the policies.

WS-I RSP default policy sets:

The Reliable Asynchronous Message Profile (WS-I RSP) default policy sets are based on the Reliable Asynchronous Message Profile specification. The WS-I RSP default policy sets include the WS-I RSP default policy set, the Lightweight Third-Party Authentication (LTPA) WS-I RSP default policy set and the Username WS-I RSP default policy set. You can use these policy sets to simplify your web services configuration.

The WS-I RSP default policy sets are composed of a set of policies to provide reliable and secure web services. The WS-I RSP default policy sets use the WS-Addressing, WS-ReliableMessaging, and WS-Security specifications. Use the WS-I RSP default policy set, the LTPA WS-I RSP default policy set, or the Username WS-Security WS-I RSP default policy set as provided with the application server. To customize the policy sets, you must first copy the policy set, and then configure custom policy settings and bindings to meet your needs.

The WS-I RSP default policy sets include the following policies:

WS-Addressing policy
: You can use the WS-Addressing policy to enable the addressing capability of the WS-Addressing specification.

WS-ReliableMessaging policy
: You can use the WS-ReliableMessaging policy to specify the quality of service for reliable delivery.

WS-Security policy
: The WS-Security policy in the WS-I RSP default policy set provides the following security:
  - Message integrity through digital signature that includes signing the body, time stamp, WS-Addressing headers and WS-ReliableMessaging headers using the WS-SecureConversation and WS-Security specifications.
  - Confidentiality through encryption that includes encrypting the body, signature elements, using the WS-SecureConversation and WS-Security specifications.
  - Traditional RSA cryptography is used to secure a request to a Trust Server to obtain a Secure Context Token (SCT). Thereafter, the conversation is secured using symmetric keys derived from the SCT.

The application server provides additional policy sets that you can use or customize. To use the following default policy sets, you must import the policy sets from the default repository. Read about importing policy sets using the administrative console for more information.

The following WS-I RSP default policy sets exist:

WS-I RSP default
This policy set provides:

- Reliable message delivery to the intended receiver by enabling WS-ReliableMessaging.
- Message integrity through digital signature that includes signing the body, time stamp, WS-Addressing headers and WS-ReliableMessaging headers using the WS-SecureConversation and WS-Security specifications.
- Confidentiality through encryption that includes encrypting the body, signature elements, using the WS-SecureConversation and WS-Security specifications.

LTPA WS-I RSP default
This policy set provides the WS-I RSP default policy set and adds a Lightweight Third Party Authentication (LTPA) token included in the request message to authenticate the client to the service.

Username WS-I RSP default
This policy set provides the WS-I RSP default policy set and adds a username token included in the request message to authenticate the client to the service. The username token is encrypted in the request.

SecureConversation default policy sets:

The SecureConversation default policy sets are based on the Web Services Secure Conversation Language (SecureConversation) standard that establishes a secure context, based on shared keys for the client and server to use for a series of messages. This standard provides a framework to define how to secure the message exchange across organizations. The SecureConversation default policy sets include the SecureConversation policy set, the Lightweight Third-Party Authentication (LTPA) SecureConversation policy set, and the Username SecureConversation policy set.

The SecureConversation default policy sets are based on the WS-SecureConversation, the WS-Security, and the WS-Addressing specifications. Use the SecureConversation policy set, the LTPA SecureConversation policy set, or the Username SecureConversation policy set as provided with the application server. To customize the policy sets, you must first copy the policy set, and then configure custom policy settings and bindings to meet your needs.

The WS-SecureConversation specification alone does not provide a complete security solution. The WS-SecureConversation is built on the WS-Security and WS-Trust specifications to provide secure communication across one or more messages. Specifically, this specification defines mechanisms for establishing and sharing security contexts, and deriving keys from established security contexts or any shared secret.

WS-Security focuses on the message authentication model but not in a security context. The WS-SecureConversation specification defines mechanisms for establishing and sharing security contexts, and deriving keys from security contexts, to enable a secure conversation. By using the SOAP extensibility model, modular SOAP-based specifications are designed to be composed with each other to provide a rich messaging environment.

The following SecureConversation default policy sets exist:

SecureConversation
This policy set provides:

- Message integrity by digital signature that includes signing the body, timestamp, and WS-Addressing headers using WS-SecureConversation and WS-Security specifications.
- Message confidentiality by encryption that includes encrypting the body, signature and signature confirmation elements, using WS-SecureConversation and WS-Security specifications.

LTPA SecureConversation
This policy set provides the SecureConversation policy set and adds a Lightweight Third Party Authentication (LTPA) token included in the request message to authenticate the client to the service.

Username SecureConversation
This policy set provides the SecureConversation policy set and adds a username token included in the request message to authenticate the client to the service. The username token is encrypted in the request

WS-ReliableMessaging default policy sets:

The WS-ReliableMessaging default policy sets are pre-configured to provide reliable message exchange between web services. Two of these policy sets (WS-I RSP and WS-I RSP ND) are immediately available, and the rest are readily available for import from a default repository.

With WS-ReliableMessaging, you can make your SOAP over HTTP-based web services reliable without writing custom code. You can use the provided non-editable default policy sets without change, or you can create customized copies of them.

All the default policy sets that include the WS-ReliableMessaging policy also include the WS-Addressing policy. The WS-ReliableMessaging policy provides the ability to deliver a message reliably to its intended receiver. The WS-Addressing policy provides a transport-neutral way to uniformly address web services and messages, and WS-ReliableMessaging uses WS-Addressing to provide asynchronous request and reply capabilities.

Note: WS-ReliableMessaging Version 1.1 messaging requires WS-Addressing to be mandatory. If you use a policy set that includes WS-ReliableMessaging and WS-Addressing policies, and the WS-Addressing policy is configured as optional, then WebSphere Application Server overrides the WS-Addressing setting and automatically enables WS-Addressing.

The following default policy sets that include the WS-ReliableMessaging policy are immediately available, as described in Viewing policy sets using the administrative console:

WS-I RSP
This policy set enables WS-ReliableMessaging Version 1.1 and uses the minimum quality of service, *unmanaged non-persistent*. This quality of service requires minimal configuration. However it is non-transactional and, although it allows for the resending of messages that are lost in the network, if a server becomes unavailable you will lose messages. This quality of service is for single server only and does not work in a cluster. In-order delivery is set to “false”, so messages are not necessarily delivered in the order in which they were sent. Message integrity is provided by digitally signing the body, the time stamp, and the WS-Addressing headers. Message confidentiality is provided by encrypting the body and the signature. This policy set follows the WS-SecureConversation and WS-Security specifications.

WS-I RSP ND
This is the network deployment version of the WS-I RSP policy set. This policy set provides the WS-I RSP default policy set and adds a *managed non-persistent* quality of service. This in-memory quality of service option uses a messaging engine to manage the sequence state, and messages are written to disk if memory is low. This quality of service allows for the re-sending of messages that are lost in the network, and can also recover from server failure. However, state is discarded after a messaging engine restart so in this case you will lose messages. This option supports clusters as well as single servers.

The following additional default policy sets that include the WS-ReliableMessaging policy are readily available for import, as described in Importing policy sets using the administrative console:

LTPA WS-I RSP
This policy set provides the WS-I RSP default policy set and adds a Lightweight Third Party Authentication (LTPA) token included in the request message to authenticate the client to the service.

Username WS-I RSP
This policy set provides the WS-I RSP default policy set and adds a username token included in the request message to authenticate the client to the service. The username token is encrypted in the request.

WSReliableMessaging 1_0
This policy set enables both WS-ReliableMessaging Version 1.0 and WS-Addressing and uses the minimum quality of service, *unmanaged non-persistent*. This quality of service requires minimal configuration. However it is non-transactional and, although it allows for the resending of messages that are lost in the network, if a server becomes unavailable you will lose messages. This quality of service is for single server only and does not work in a cluster. In-order delivery is set to “false”, so messages are not necessarily delivered in the order in which they were sent.

You can use this policy set with .NET-based web services.

WSReliableMessaging default
This policy set enables both WS-ReliableMessaging Version 1.1 and WS-Addressing and uses the minimum quality of service, *unmanaged non-persistent*. This quality of service requires minimal configuration. However it is non-transactional and, although it allows for the resending of messages that are lost in the network, if a server becomes unavailable you will lose messages. This quality of service is for single server only and does not work in a cluster. In-order delivery is set to “false”, so messages are not necessarily delivered in the order in which they were sent.

WSReliableMessaging persistent
This policy set enables both WS-ReliableMessaging and WS-Addressing and uses the maximum quality of service, *managed persistent*. This quality of service supports asynchronous web service invocations and uses a service integration messaging engine and message store to manage the sequence state. Messages are processed within transactions, are persisted at the web service requester server and at the web service provider server, and are recoverable in the event of server failure. In-order delivery is set to “false”, so messages are not necessarily delivered in the order in which they were sent.

Because this policy set specifies managed persistent quality of service, you have to define bindings to the service integration bus and messaging engine that you want to use to manage the WS-ReliableMessaging state. You can attach and bind a WS-ReliableMessaging policy set to a web service application by using the administrative console or the wsadmin tool.

WSAddressing default policy set:

The WSAddressing default policy set provides a transport-neutral way to uniformly address web services and messages.

The WSAddressing default policy set is based on the WS-Addressing specification. The WS-Addressing standard uses endpoint references and message addressing properties to facilitate the addressing of web services in a standard and interoperable way.

Use the WSAddressing default policy set as provided with the application server. To customize the policy set, you must first copy the policy set, and then configure custom policy settings and bindings to meet your needs.

To learn more about the WS-Addressing standard, read about Web Services Addressing support.

Web Services Security default policy sets:

The Web Services Security default policy sets are based on the WS-Security 1.0 and Web Services Addressing (WS-Addressing) specifications. The Web Services Security default policy sets include the WSSecurity default policy set, the Lightweight Third-Party Authentication (LTPA) WSSecurity policy set, the Username WSSecurity policy set, and the Kerberos V5 HTTPS default policy set. These default policy sets are used to build secure web services.

The Web Services Security default policy sets use the WS-Security 1.0 specification enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. Providing quality of protection means to prevent the following potential threats to SOAP messages:

- The message being modified or read by antagonists.
- An antagonist sending messages to a service that are formed correctly, but lack the appropriate security claims to be processed.

The WS-Addressing specification defines XML 1.0 and XML Namespaces elements to identify web services endpoints and to secure end-to-end endpoint identification in messages.

You can use the WSSecurity default policy set, the LTPA WSSecurity policy set, the Username WSSecurity policy set, or the Kerberos V5 HTTPS default policy set as provided with the application server. To customize the other Web Services Security policy sets, you must first copy the policy set, and then configure custom policy settings and bindings to meet your needs.

Features and details of the default Web Services Security policy sets are as follows:

Kerberos V5 HTTPS default

This policy set provides message authentication with a Kerberos Version 5 token. Message integrity and confidentiality are provided by Secure Sockets Layer (SSL) transport security. This policy set follows the OASIS Kerberos Token Profile V1.1 and WS-Security specifications.

When you use this policy set, configure the basic authentication data and custom properties such as the com.ibm.wsspi.wssecurity.krbtoken.targetServiceName and com.ibm.wsspi.wssecurity.krbtoken.targetServiceHost custom properties in the client bindings. For more information, see the Authentication generator or consumer token settings and Protection token settings (generator or consumer) topics.

LTPA WSSecurity default

This policy set provides:

- Message integrity through digital signature (using RSA public-key cryptography) to sign the body, time stamp, and WS-Addressing headers using WS-Security specifications.
- Message confidentiality through encryption (using RSA public-key cryptography) to encrypt the body, signature and signature elements using WS-Security specifications.
- A Lightweight Third Party Authentication (LTPA) token included in the request message to authenticate the client to the service.

Username SecureConversation

This policy set provides:

- Message integrity through digital signature that includes signing the body, time stamp, and WS-Addressing headers using WS-SecureConversation and WS-Security specifications
- Message confidentiality through encryption that includes encrypting the body, signature and signature confirmation elements, using WS-SecureConversation and WS-Security specifications
- A username token included in the request message to authenticate the client to the service. The username token is encrypted in the request

Username WSSecurity default

This policy set provides:

- Message integrity through digital signature (using RSA public-key cryptography) to sign the body, time stamp, and WS-Addressing headers using WS-Security specifications.
- Message confidentiality through encryption (using RSA public-key cryptography) to encrypt the body, signature and signature elements using WS-Security specifications.
- A username token included in the request message to authenticate the client to the service. The username token is encrypted in the request.

WSSecurity default

This policy set provides:

- Message integrity through digital signature (using RSA public-key cryptography) to sign the body, time stamp, and WS-Addressing headers using WS-Security specifications.
- Message confidentiality through encryption (using RSA public-key cryptography) to encrypt the body, signature and signature elements using WS-Security specifications.

WSTransaction default policy sets:

The WSTransaction default policy sets are based on the WS-Transaction specification and provide transactional integrity for web services. The WSTransaction default policy sets include the WSTransaction policy set and the SSL WSTransaction policy set.

You can use the WSTransaction default policy sets to make your SOAP over HTTP-based web services interoperable and coordinate atomic transactions or business activities without writing custom code. Use the WSTransaction policy set or the SSL WSTransaction policy set as provided with the application server. To customize the policy sets, you must first copy the policy set, and then configure custom policy settings and bindings to meet your needs.

The WSTransaction default policy sets are:

WSTransaction

Use this policy set to coordinate distributed transactional work atomically and interoperably, by using the WS-AtomicTransaction specification. Also, use this policy set to coordinate loosely coupled business processes that are distributed across the heterogenous web service environment, with the ability to compensate actions if a failure occurs in the business activity, by using the WS-BusinessActivity specification.

SSL WSTransaction

Use this policy set to coordinate distributed transactional work atomically, interoperably and securely, by using the WS-AtomicTransaction specification and SSL Transport security. Also, use this policy set to coordinate loosely coupled business processes, with the ability to compensate actions if a failure occurs in the business activity, securely, by using the WS-BusinessActivity specification and SSL Transport security.

WSHTTPS default policy set:

The WSHTTPS default policy set provides SSL transport security for the HTTP protocol with web services applications.

The WSHTTPS default policy set is provided with the application server and it contains the HTTP transport policy, the SSL transport policy and the WS-Addressing policy.

You can use the WSHTTPS default policy set as provided with the application server. You cannot edit the WSHTTPS default policy set. You can create a copy of the default policy set and then configure custom policy settings and bindings to meet your needs. Alternatively, you can create a new policy set and specify the policies for it.

Copy of default policy set and bindings settings:

Use this page to copy a policy set that you select from a list of available policy sets.

To view this administrative console page:

1. Click **Services > Policy sets > Application policy sets**.
2. Select the policy set that you want to copy, and click **Copy**.
3. Enter a name for the copy of the policy set in the **Name** field.
4. [Optional] Enter a description for the copy of the policy set in the **Description** field.

Name:

Specifies the name of the policy set. Use this field to enter a name for the copy of the policy set.

Description:

Specifies a description of the policy set that you want to copy.

Transfer attachments:

If the policy set that you want to copy is attached to one or more applications, services, or endpoints, then the check box option for **Attach this policy set in place of the original for all attached applications, services, endpoints, and operations**, is available. The default setting for this check box is cleared. You can perform the following actions:

1. Select the **Attach this policy set in place of the original for all attached applications, services, endpoints, and operations** check box to move the attachments to a new policy set. Selecting the check box detaches the original policy set and attaches the replacement policy set in its place.
2. Select **Copy bindings** if you want to copy the bindings of the policy set that is currently attached.
3. [Optional] Select **Restore default bindings** if you want to restore the default bindings.

Configuring the WS-Addressing policy

When working with policy sets in the administrative console, you can add and configure policies to enable standard addressing of web services.

Before you begin

You can specify policies for custom policy sets. The provided default policy sets cannot be edited. You must create a copy of the default policy set or create a completely new policy set in order to specify the policies for it.

About this task

Adding a WS-Addressing policy enables the support for WS-Addressing. This support provides a standard way to address Web services and include addressing information in messages. Adding a WS-Addressing policy is equivalent to configuring the WSDL file for the web service to specify that WS-Addressing should be used.

To specify or configure the policies associated with a policy set, use the administrative console.

Procedure

1. In the navigation pane of the administrative console, click **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name*** > **[Policies] WS-Addressing**. The WS-Addressing settings pane is displayed.

2. Select **WS-Addressing is mandatory** to specify that WS-Addressing information must be included in SOAP message headers. For servers, this setting means that the server returns a fault if it receives a message that does not contain a WS-Addressing header. For clients, this setting means that WS-Addressing headers are always added to SOAP messages. If you have enabled WS-Policy, this requirement is communicated between servers and clients that support WS-Policy.
3. In the **Messaging style** box, select the message exchange pattern to use:
   - **Synchronous and asynchronous**. The targeting of response messages is not restricted.
   - **Synchronous only**. Response messages must be targeted at the WS-Addressing anonymous URI.
   - **Asynchronous only**. Response messages must not be targeted at the WS-Addressing anonymous URI.
4. Click **OK**.
5. Save your changes to the master configuration.

Results

After you have included the WS-Addressing policy in a policy set, the associated policy set uses this policy to address web services.

WS-Addressing policy settings:

Use this page to define the appropriate WS-Addressing policy assertions for this policy set.

To view this administrative console page, click **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name*** > **[Policy] WS-Addressing**, when the policy set includes the WS-Addressing policy type.

You can configure the WS-Addressing policy type for both client-side and provider-side policy sets. If you enable WS-Policy, this configuration is communicated between servers and clients that support WS-Policy.

WS-Addressing is mandatory:

Specifies whether a WS-Addressing SOAP header is included on messages.

Information	**Value**
Data type	Check box
Default	Cleared
Range	**Cleared** WS-Addressing is not mandatory. Servers will not generate a fault if they receive a message that does not contain a WS-Addressing header. Clients might not include WS-Addressing headers in SOAP messages, for example, if WS-Policy is enabled and the server does not specify that WS-Addressing is mandatory. **Selected** WS-Addressing is mandatory. Servers return a fault if they receive a message that does not contain a WS-Addressing header. Clients always include WS-Addressing headers in SOAP messages.

Messaging style:

Specifies the messaging style supported by this policy set.

Use the radio buttons to configure the messaging style.

- Select **Synchronous and asynchronous** to specify that there is no restriction on the targeting of response messages.
- Select **Synchronous only** to specify that response messages must be targeted at the WS-Addressing anonymous URI. You might want to use this messaging style in the following situations:
  - The SOAP headers are not signed, and WS-Security is not enabled. Specifying the synchronous message exchange pattern prevents the server sending messages to a third party, thereby preventing the server from potentially taking part in a Denial of Service attack.
  - Clients with a NAT device between themselves and the endpoint. In this configuration, non-anonymous URIs cannot be routed. Use this setting to prevent the client from sending a message containing a ReplyTo endpoint reference with a non-anonymous URI.
- Select **Asynchronous only** to specify that response messages must not be targeted at the WS-Addressing anonymous URI. This setting does not mean that all non-anonymous URIs are supported, therefore a server can return a fault if it receives a response endpoint reference that it cannot process. You might want to use this messaging style if the endpoint has a very long-running invocation time, and you do not want to hold the connection open while waiting for a response.

The following table shows how the messaging style options correspond to WS-Policy assertions.

Table 140. WS-Addressing messaging style and WS-Policy. This table provides a mapping between messaging style options and WS-Policy assertions.

Messaging style	WS-Policy mapping
Synchronous and asynchronous	wsam:AnonymousResponses or wsam:NonAnonymousResponses
Synchronous only	wsam:AnonymousResponses
Asynchronous only	wsam:NonAnonymousResponses

Information	Value
Required	No
Data type	Radio button

Attaching a policy set to a service artifact

Attach a policy set to a service artifact, such as an application, service, endpoint or operation, to define the quality of services that are supported. Policy sets can define the policies for WS-Addressing, WS-Security, WS-ReliableMessaging, WS-Transaction, HTTP transport, Java Messaging Service (JMS) transport, and Secure Sockets Layer (SSL) transport.

Before you begin

Before you can start this task, you must deploy an application containing web services. Also, if none of the default policy sets contain the necessary policy definitions, then you must create a custom policy set with the necessary definitions.

About this task

Develop a web service that contains each of the necessary artifacts, and deploy your web services application into your application server instance. Now you can attach policy sets to your service artifacts, such as an application, service, or endpoint.

To attach a policy set to a service artifact, perform the following steps:

Procedure

1. Open the administrative console.

2. To attach a policy set to a service provider, click **Applications > Enterprise applications** > ***application_name*** > **Service provider policy sets and bindings**.

   To attach a policy set to a service client, click **Applications > Enterprise applications** > ***application_name*** > **Service client policy sets and bindings**.
3. Select the check box for the service artifact.
4. Select **Attach** to display a list of available policy sets to attach. Select a policy set from the list.
5. Click **Save**, to save your changes to the master configuration.
6. [Optional] To see what attachments are defined for a given policy set, select **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name*** > **Attached applications**.

Results

When you finish these steps, a policy set is attached to the service artifact.

Example

If you have the application, `app1` and you want to attach the policy set, `WSSecurity default`, then perform the following steps:

1. Locate the `app1` application in the **Applications** > **Enterprise applications** collection.
2. Click the **app1** application.
3. Click the **Service provider policy sets and bindings** link or the **Service client policy sets and bindings** link.
4. Select the check box for the service artifact where the policy set is to be attached.
5. Click **Attach**. Select **WSSecurity default** policy set.
6. Click **Save**, to save your changes to the master configuration.

What to do next

You can create custom bindings for policy set attachments. Read about creating custom bindings for policy set attachments.

You can configure the service client or service provider to share their policies. Read about using WS-Policy to exchange policies in a standard format.

Configuring a service provider to share its policy configuration

A WebSphere Application Server service provider can share its policy configuration in published Web Services Description Language (WSDL), or WSDL that is obtained by using an HTTP GET request or the Web Services Metadata Exchange (WS-MetadataExchange) GetMetadata request.

Before you begin

You have developed a web services service provider that contains all the necessary artifacts and deployed your web services application into your application server instance. You have attached the policy sets and managed the associated bindings.

For a list of WS-Policy assertion specifications and WS-Policy domains that are supported, see the topic about learning about WS-Policy.

About this task

You can make the policy configuration of a Java API for XML-Based Web Services (JAX-WS) service endpoint available to share in the following ways:

- Include the policy configuration of the service provider in the WSDL. The WSDL is then available to publish, or to obtain by using an HTTP GET request.
- Enable the Web Services Metadata Exchange (WS-MetadataExchange) protocol so that the policy configuration of the service provider is included in the WSDL and is available to a WS-MetadataExchange GetMetadata request. An advantage of using the WS-MetadataExchange protocol is that you can apply message-level security to WS-MetadataExchange GetMetadata requests by using a suitable system policy set.

If the service provider application uses multipart WSDL, all the WSDL must be local to the web service application. For more information about multipart WSDL, see the topic about WSDL.

You must configure a service provider to share its policy configuration because by default the policy configuration is not available in its WSDL. You can configure the service provider to include the policy configuration in its WSDL, to use WS-MetadataExchange so that the policy configuration is available, or both. This topic describes how to configure a service provider to share its policy configuration by using the administrative console. You can also configure a service provider to share its policy configuration by using wsadmin commands or Rational Application Developer tools.

You can configure a service provider to share its policy configuration at application or service level. The policy configuration that is represented by the policy sets attached to any lower levels will also be shared. Policy sets that are attached at lower levels override the policy set configuration attached at a higher level.

Procedure

1. From the navigation pane of the administrative console, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***service_provider_application_instance*** > **[Web services properties] Service provider policy sets and bindings**.
2. In the row for the application or service where the provider policy that you want to share is attached, click the link in the Policy sharing column. The link is either **Enabled** or **Disabled**. The Policy Sharing pane is displayed.
3. To include the policy configuration of the service provider in its WSDL so that it can be either published or obtained by using an HTTP GET request, select **Exported WSDL**.
4. To enable WS-MetadataExchange and make the policy configuration of the service provider available to a WS-MetadataExchange GetMetada request, select **WS-MetadataExchange request**.
5. Optional: If you select **WS-MetadataExchange request** and you want to use message-level security, select **Attach a system policy set to the WS-MetadataExchange**, then select a suitable policy set and binding from the drop-down lists. See Configuring security for a WS-MetadataExchange request.
6. Click **OK** and save your changes to the master configuration.

Results

The policy configuration of the service provider is available to its clients. The WSDL of the service provider contains the current policy configuration in WS-PolicyAttachments format so that it is available to other clients, service registries, or services that support the Web Services Policy (WS-Policy) specification. The link in the Policy Sharing column on the Service provider policy sets and bindings pane changes to **Enabled**.

If the policy configuration cannot be shared, an error that describes the problem is written to the service provider error log, and the following policy is attached to the WSDL of the service provider:

```
<wsp:Policy>
<wsp:ExactlyOne>
</wsp:ExactlyOne>
</wsp:Policy>
```

This policy notifies the client that there is no acceptable policy configuration for the service. Other aspects of the WSDL are unaffected.

A service provider might not be able to share its policy configuration because the configuration cannot be expressed in the standard WS-PolicyAttachments format. One reason might be because multiple incompatible policies are defined for a particular attach point. Another reason might be because there is not enough binding information to generate the standard policy. Policy configuration might include bootstrap policy, for example, the policy to access a WS-Trust service, so the bootstrap policy must also be expressed in WS-PolicyAttachments format.

Configuring a service provider to share its policy configuration using wsadmin scripting:

A WebSphere Application Server service provider can share its policy configuration in published Web Services Description Language (WSDL), or WSDL that is obtained by using an HTTP GET request or the Web Services Metadata Exchange (WS-MetadataExchange) GetMetadata request.

Before you begin

You have developed a web services service provider that contains all the necessary artifacts and deployed your web services application into your application server instance. You have attached the policy sets and managed the associated bindings.

For a list of WS-Policy assertion specifications and WS-Policy domains that are supported, see the topic about learning about WS-Policy.

About this task

You can make the policy configuration of a Java API for XML-Based Web Services (JAX-WS) service endpoint available to share in the following ways:

- Include the policy configuration of the service provider in the WSDL. The WSDL is then available to publish, or to obtain by using an HTTP GET request.
- Enable the Web Services Metadata Exchange (WS-MetadataExchange) protocol so that the policy configuration of the service provider is included in the WSDL and is available to a WS-MetadataExchange GetMetadata request. An advantage of using the WS-MetadataExchange protocol is that you can apply message-level security to WS-MetadataExchange GetMetadata requests by using a suitable system policy set.

If the service provider application uses multipart WSDL, all the WSDL must be local to the web service application. For more information about multipart WSDL, see the topic about WSDL.

You must configure a service provider to share its policy configuration because by default the policy configuration is not available in its WSDL. You can configure the service provider to include the policy configuration in its WSDL, to use WS-MetadataExchange so that the policy configuration is available, or both. This topic describes how to configure a service provider to share its policy configuration by using wsadmin commands. You can also use the administrative console or Rational Application Developer tools.

You can configure a service provider to share its policy configuration at application or service level. The policy configuration that is represented by the policy sets attached to any lower levels will also be shared. Policy sets that are attached at lower levels override the policy set configuration attached at a higher level.

Procedure

1. Start the wsadmin scripting client if it is not already running.
2. Use the **SetProviderPolicySharingInfo** command. For example:

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WebServiceProviderApplication
-resource WebService:/WebServiceProvider.war:{http://example_path/}Service1
-sharePolicyMethods [httpGet ]]')
```

3. Save your changes to the master configuration.

   To save your configuration changes, enter the following command:

```
AdminConfig.save()
```

Results

The policy configuration of the service provider is available to its clients. The WSDL of the service provider contains the current policy configuration in WS-PolicyAttachments format so that it is available to other clients, service registries, or services that support the Web Services Policy (WS-Policy) specification.

If the policy configuration cannot be shared, an error that describes the problem is written to the service provider error log, and the following policy is attached to the WSDL of the service provider:

```
<wsp:Policy>
<wsp:ExactlyOne>
</wsp:ExactlyOne>
</wsp:Policy>
```

This policy notifies the client that there is no acceptable policy configuration for the service. Other aspects of the WSDL are unaffected.

A service provider might not be able to share its policy configuration because the configuration cannot be expressed in the standard WS-PolicyAttachments format. One reason might be because multiple incompatible policies are defined for a particular attach point. Another reason might be because there is not enough binding information to generate the standard policy. Policy configuration might include bootstrap policy, for example, the policy to access a WS-Trust service, so the bootstrap policy must also be expressed in WS-PolicyAttachments format.

What to do next

Optionally, you can publish the WSDL files.

setProviderPolicySharingInfo command:

Use the **setProviderPolicySharingInfo** command to set how an application or service that is a web service provider can share its policy configuration with other clients, service registries, or services that support the WS-Policy specification. You can set or remove this information about how a provider policy is shared.

To run the command, use the AdminTask object of the wsadmin scripting client.

This command is valid only when it is used with WebSphere Application Server Version 7 and later application servers. Do not use it with earlier versions.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using the command, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

Purpose

Use the **setProviderPolicySharingInfo** command to set how an application, or a service in an application, shares its policy configuration with clients, service registries, or services that support the WS-Policy specification. The policy configuration is shared in WS-PolicyAttachments format.

The policy configuration of the resource can be shared with clients through a WS-MetadataExchange request, through Web Services Description Language (WSDL) exported by a `?WSDL` HTTP Get request, or through both methods.

Target object

None.

Required parameters

-applicationName
The name of the application for which you want to set how the provider policy is shared. (String)

-resource
The name of the resource for which you want to set how the provider policy is shared. For all resources in an application, specify `WebService:/`. For a service in an application, specify `WebService:/`*`module`*`:{`*`namespace`*`}`*`service_name`*. Endpoints or operations inherit the settings of the parent application or service. (String)

Optional parameters

-sharePolicyMethods
Specifies how the policy configuration of the resource can be shared. (String array)

Enter either or both of the following values:

httpGet
The resource can share its policy configuration through WSDL that is obtained by a `?WSDL` HTTP Get request.

wsMex The resource can share its policy configuration through a WS-MetadataExchange request.

-wsMexProperties
Specifies that message-level security is required for WS-MetadataExchange requests and specifies the settings that provide the message-level security. (Properties)

Enter the following values, following each value with the setting that you require for that value:

wsMexPolicySetName
The name of the system policy set that specifies message-level security when the resource shares its policy configuration through a WS-MetadataExchange request. Specify a system policy set that contains only WS-Security policies, only WS-Addressing policies, or both. The default policy set is `SystemWSSecurityDefault`.

wsMexPolicySetBinding
The name of the general binding for the policy set attachment when the resource shares its policy configuration through a WS-MetadataExchange request. Specify a general binding that is scoped to the global domain, or scoped to the security domain of this service. If you do not specify this property, the default binding is used.

This parameter is valid only when you specify `wsMex` for the **sharePolicyMethods** parameter.

-remove
Specifies whether the information about how the provider policy is shared is removed from the resource. (Boolean)

This parameter takes the following values:

true The information about how the provider policy is shared is removed from the resource.

false This value is the default. The information about how the provider policy is shared is not removed from the resource.

Examples

The following example removes the information about how the provider policy is shared from the WSSampleServices application:

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WSSampleServices
-resource WebService:/ -remove true]')
```

The following example enables policy sharing, using WSDL exported by a `?WSDL` HTTP Get request, for the EchoService service in the WSSampleServices application:

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WSSampleServices
-resource WebService:/WSSampleServicesSei.war:{http://example_path/}EchoService
-sharePolicyMethods [httpGet ]]')
```

The following example enables policy sharing, using a WS-MetadataExchange request with message-level security, for the WSSampleServices application. Message level security is provided by the SystemWSSecurityDefault policy set and the “Provider sample” general binding.

```
AdminTask.setProviderPolicySharingInfo('[-applicationName WSSampleServices
-resource WebService:/ -sharePolicyMethods [wsMex ]
-wsMexProperties [ [wsMexPolicySetName [SystemWSSecurityDefault]]
[wsMexPolicySetBinding [Provider sample]] ]]')
```

getProviderPolicySharingInfo command:

Use the **getProviderPolicySharingInfo** command to find out whether an application or service that is a web service provider can share its policy configuration, and list the properties that apply to sharing that configuration.

To run the command, use the AdminTask object of the wsadmin scripting client.

This command is valid only when it is used with WebSphere Application Server Version 7 and later application servers. Do not use it with earlier versions.

For a list of the available policy set management administrative commands, plus a brief description of each command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('PolicySetManagement')
```

For overview help on a given command, enter the following command at the wsadmin prompt:

```
print AdminTask.help('command_name')
```

After using the command, save your changes to the master configuration. For example, use the following command:

```
AdminConfig.save()
```

Purpose

Use the **getProviderPolicySharingInfo** command to find out how a web services application, or a service in a Web services application, shares its policy configuration with clients, service registries, or services that support the WS-Policy specification. The policy configuration is shared in WS-PolicyAttachments format.

The command returns properties that show whether the policy configuration of the resource can be shared with clients through a WS-MetadataExchange request or through Web Services Description Language (WSDL) that is obtained by a `?WSDL` HTTP Get request.

Target object

None.

Required parameters

`-applicationName`
The name of the application for which you want to find out how it shares its policy configuration. The application must be a service provider. (String)

Optional parameters

`-resource`
The name of the resource for which you want to find out how it shares its policy configuration. If you specify this parameter, only the properties for that resource are returned. To retrieve information for the application, specify `WebService:/`. Alternatively, you can specify a service, endpoint or operation. However, policy sets are attached only at the application or service level, so the properties returned for an endpoint or operation are the settings that are inherited from the service. (String)

Return value

Returns a list of properties that include the resource name and that show whether the policy configuration of the resource can be shared. The following properties can be returned:

`wsMexPolicySetName`
The name of the policy set that specifies message-level security when the resource shares its policy configuration through a WS-MetadataExchange request. This property is returned if the value of the **`sharePolicyMethods`** property is `wsMex` and a policy set to provide message-level security was specified.

`wsMexPolicySetBinding`
The name of the binding that is applied when the resource shares its policy configuration through a WS-MetadataExchange request. This property is returned if the value of the **`sharePolicyMethods`** property is `wsMex` and a binding to provide message-level security was specified.

`resource`
The resource that you specified.

`directSetting`
How the properties apply to the resource. Valid values for this property are:

`true`
The properties apply directly to the resource.

`false`
The properties are inherited from the parent application or service.

`sharePolicyMethods`
How the policy configuration of the resource can be shared. Valid values for this property are:

`httpGet`
The resource shares its policy configuration through an HTTP Get request.

`wsMex`
The resource shares its policy configuration through a WS-MetadataExchange request.

Example

The following command displays the policy sharing configuration properties for the EchoService service in the WSSampleServices application. The provider is configured to share its policy through an HTTP Get request, and a WS-MetadataExchange request with message-level security. Message-level security for the WS-MetadataExchange request is provided by using the SystemWSSecurityDefault policy set and the "Provider sample" general binding.

```
AdminTask.getProviderPolicySharingInfo(['-applicationName', 'WSSampleServices',
'-resource', 'WebService:/SampleServicesSei.war:{http://example_path/}EchoService'])
.
.
[ [wsMexPolicySetName SystemWSSecurityDefault] [wsMexPolicySetBinding [Provider sample]]
[resource WebService:/SampleServicesSei.war:{http://example_path/}EchoService/]
[directSetting true] [sharePolicyMethods [httpGet wsMex]] ]
```

Policy sharing settings:

Use this pane to view and change whether the policy configuration of a web services service provider is shared. You can configure the service provider to include the policy configuration in its Web Services Description Language (WSDL) so that it can be accessed using an HTTP Get request, or published. You can also make the policy configuration available to a Web Services Metadata Exchange (WS-MetadataExchange) request.

To view this administrative console page for an application or service, complete the following steps. The application must be a web services service provider.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***service_provider_application_name*** > **[Web Services Properties] Service provider policy sets and bindings**.
2. Select the link in the Policy Sharing column for the application or service you require. The link is available if the application or service has a policy set attached.

To view this administrative console page for a service, complete the following steps.

1. Click **Services** > **Service providers** > ***service_name***.
2. Select the link in the Policy Sharing column for the service. The link is available if the service has a policy set attached.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Exported WSDL:

Select **Exported WSDL** to include the policy configuration of the service provider in the WSDL. The policy configuration is in WS-PolicyAttachments format in the WSDL so that it is then available to other clients, service registries, or services that support the Web Services Policy (WS-Policy) specification. The policy configuration will be available in published WSDL, or a client can use an HTTP Get request that is targeted at the target URL followed by `?WSDL` to obtain the provider policy.

WS-MetadataExchange request:

Select **WS-MetadataExchange** to make the policy configuration of the service provider available to a WS-MetadataExchange GetMetadata request. The policy configuration is in WS-PolicyAttachments format in the WSDL so that it is then available to other clients, service registries or services that support the Web Services Policy (WS-Policy) specification and the WS-MetadataExchange GetMetadata request.

When **WS-MetadataExchange** is selected, the **Attach a system policy set to the WS-MetadataExchange** option is available.

Attach a system policy set to the WS-MetadataExchange:

Select **Attach a system policy set to the WS-MetadataExchange** to set message-level security for the WS-MetadataExchange request. By default, this option is not selected and the transport policy of the application is used. This option is available only when **WS-MetadataExchange** is selected.

When **Attach a system policy set to the WS-MetadataExchange** is selected, the **Policy set** and **Binding** lists are available.

Policy set:

Select the policy set you require from the list to provide message-level security for the WS-MetadataExchange request. You can select from system policy sets that contain only WS-Security policies, only WS-Addressing policies, or both. The default policy set is SystemWSSecurityDefault.

System policy sets are used for system messages that are not business-related, for example, messages that apply qualities of service (QoS), including the messages that are defined in the WS-MetadataExchange protocol.

Note that any transport policy of the application is always used.

This option is available only when **Attach a system policy set to the WS-MetadataExchange** is selected.

Binding:

Select the binding you require from the list to provide message-level security for the WS-MetadataExchange request. You can select from general bindings that are scoped to the global domain, or scoped to the security domain of this service.

This option is available only when **Attach a system policy set to the WS-MetadataExchange** is selected.

Configuring the client policy to use a service provider policy

An application that is a web service client can obtain the policy configuration of a web service provider and use this information to establish a policy configuration that is acceptable to both the client and the service provider.

Before you begin

You have developed a web service client that contains all the necessary artifacts, and deployed your web services application into your application server instance. If you require them, you have attached the policy sets and managed the associated bindings.

The service provider must publish its policy in its Web Services Description Language (WSDL) and that policy must contain its policy configuration at run time in WS-PolicyAttachments format. The client must be able to support those provider policies.

For a list of WS-Policy assertion specifications and WS-Policy domains that are supported, see the WS-Policy topic.

About this task

You can administer the client to configure itself dynamically at run time, based on the policy of the service provider in the standard WS-PolicyAttachments format. You can administer the client to apply dynamically the provider policy at the application or service or service reference level. By default, endpoints and

operations inherit their policy configuration from the relevant service. However, it is possible to configure a service reference to override the service, in which case the endpoints and operations inherit their policy configuration from the service reference.

If the provider policy uses multipart WSDL, you can use an HTTP GET request to obtain the policy of the provider, but you cannot use the WS-MetadataExchange protocol. For more information about multipart WSDL, see the topic about WSDL.

Policy intersection is the comparison of a client policy and a provider policy to determine whether they are compatible, and the calculation of a new policy, known as the effective policy, that complies with both their requirements and capabilities.

This topic describes how to configure the client policy to use a service provider policy by using the administrative console. You can also configure the client policy to use a service provider policy by using wsadmin commands.

Procedure

1. From the navigation panel of the administrative console, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***service_client_application_instance*** > **[Web services properties] Service client policy sets and bindings**.
2. In the row for the application or service where you want to apply the policy, click the link in the Policies Applied column. The Policies Applied panel is displayed.
3. Select one of the following options from the drop-down list:
   - Provider policy only. Configure the client based solely on the policy of the service provider. This option is available when a client policy set is not attached.
   - Client and provider policy. Configure the client based on both the client policy set and the policy of the service provider. This option is available when a client policy set is attached.

   The other options in the list do not apply to this task.
4. Use the radio buttons to select which method to employ to obtain the provider policy: an HTTP GET request (see step 5) or a WS-MetadataExchange request (see step 6).
5. Optional: To obtain the provider policy by using an HTTP GET request, click **HTTP GET request**. By default, the HTTP GET request is targeted at the URL for the service endpoint followed by `?WSDL`. For example:

```
http://myhost:9080/WSSampleSei/EchoService?WSDL
```

   When the policy set attach point is at the application level you cannot change this value.
   a. Optional: If you are applying a policy to a service and the provider policy is located at the service endpoint, ensure that **Use the default request target** is selected.
   b. Optional: If you are applying a policy to a service and the provider policy is not located at the service endpoint, click **Specify request target**, then enter the URL for the location of the provider policy in the field. For example, you might change the target of the HTTP GET request if the provider policy is located in a repository.
   c. Optional: If you select **HTTP GET request** as the method to be used to obtain the provider policy and if you select **Specify request target** and you want to configure transport-level security, select **Attach a system policy set to the HTTP GET request**, then select a suitable policy set and binding from the drop-down lists. Select the policy set you require from the Policy set list to provide transport-level security for the HTTP GET request. Select from system policy sets that contain solely HTTP transport policies, solely SSL transport policies, or both; the policy set cannot contain other policy types. Select the binding you require from the Binding list for the HTTP GET request. You can select from general bindings that are scoped to the global domain or scoped to the security domain of this service.
6. Optional: To obtain the provider policy by using a Web Services Metadata Exchange (WS-MetadataExchange) GetMetadata request, click **WS-MetadataExchange request**.

a. Optional: If you select **WS-MetadataExchange request** and want to use message-level security, select **Attach a system policy set to the WS-MetadataExchange request**, then select a suitable policy set and binding from the drop-down lists. See Configuring security for a WS-MetadataExchange request.

7. Click **OK**.
8. Save your changes to the master configuration.

Results

The web application client-side policy is calculated when it is required at run time, based either on the policy of the service provider, or on the client policy set and the policy of the service provider, depending on which option you selected. This calculated policy is known as the “effective policy” and is cached as a runtime configuration. The effective policy is used for subsequent outbound web service requests to the endpoint or operation for which the dynamic policy calculation was performed. The policy set configuration of the client does not change.

The provider policy that the client holds for a service is refreshed the first time that the web service is invoked after the application is loaded. After that, the provider policy is refreshed when the application restarts, or if the application explicitly invokes a refresh. When the provider policy is refreshed, the effective policy is recalculated.

Configuring the client policy to use a service provider policy by using wsadmin scripting:

An application that is a web service client can obtain the policy configuration of a web service provider and use this information to establish a policy configuration that is acceptable to both the client and the service provider.

Before you begin

You have developed a web service client that contains all the necessary artifacts, and deployed your web services application into your application server instance. If you require them, you have attached the policy sets and managed the associated bindings.

The service provider must publish its policy in its Web Services Description Language (WSDL) and that policy must contain its policy configuration at run time in WS-PolicyAttachments format. The client must be able to support those provider policies.

For a list of WS-Policy assertion specifications and WS-Policy domains that are supported, see the WS-Policy topic.

About this task

You can administer the client to configure itself dynamically at run time, based on the policy of the service provider in the standard WS-PolicyAttachments format. You can administer the client to dynamically apply the provider policy at the application or service or service reference level.

Note: If you specify client dynamic policy control at the service reference level, you must use the new name-value paired list format of the resource string. If you are not specifying client dynamic policy control at service reference level, you can use either format.

Table 141. How to specify policy control at different levels of the application. For each applicable level of the application, the table lists the relevant string format command and name-value pair format command needed to specify policy control and summarizes the associated behavior.

Level	String format	Name-value pair list format (NEW)	Behavior
Type	`"WebService:/"`	`"type=WebService:/"`	Indicates all artifacts in the application

Table 141. How to specify policy control at different levels of the application (continued). For each applicable level of the application, the table lists the relevant string format command and name-value pair format command needed to specify policy control and summarizes the associated behavior.

Level	String format	Name-value pair list format (NEW)	Behavior
Service	"WebService:/*myModule*:{*namespace*}*myService*"	"type=WebService:/,module=*myModule*,service={*namespace*}*myService*"	Indicates all artifacts within the web service
Endpoint (under this service)	"WebService:/*myModule*:{*namespace*}*myService*/*endpointA*"	"type=WebService:/ ,module=*myModule*,service={*namespace*}*myService*,endpoint=*endpointA*"	Indicates all operations for this endpoint (under the service)
Operation (under this service)	"WebService:/*myModule*:{*namespace*}*myService*/*endpointA*/*operation1*"	"type=WebService:/ ,module=*myModule*,service={*namespace*}*myService*,endpoint=*endpointA*, operation=*operation1*"	Indicates a specific single operation (under the service)
Service reference	[Not possible]	"type=WebService:/ ,module=*myModule*,service={*namespace*}*myService*,serviceRef=*myServiceRef*"	Indicates all artifacts within the web service reference
Endpoint (under this service reference)	[Not possible]	"type=WebService:/ ,module=*myModule*,service={*namespace*}*myService*,serviceRef=*myServiceRef*, endpoint=*endpointA*"	Indicates all operations for this endpoint (under the service reference)
Operation (under this service reference)	[Not possible]	"type=WebService:/ ,module=*myModule*,service={*namespace*}*myService*,serviceRef=*myServiceRef*, endpoint=*endpointA*,operation=*operation1*"	Indicates a specific single operation (under the service reference)

If the provider policy uses multipart WSDL, you can use an HTTP GET request to obtain the policy of the provider, but you cannot use the WS-MetadataExchange protocol. For more information about multipart WSDL, see the topic about WSDL.

Policy intersection is the comparison of a client policy and a provider policy to determine whether they are compatible, and the calculation of a new policy, known as the effective policy, that complies with both their requirements and capabilities.

This topic describes how to configure the client policy to use a service provider policy by using wsadmin commands. You can also configure the client policy to use a service provider policy by using the administrative console.

Procedure

1. Start the wsadmin scripting client if it is not already running.
2. Use the **SetClientDynamicPolicyControl** command. For example:

```
AdminTask.setClientDynamicPolicyControl('[-applicationName WebServiceClientApplication
-resource WebService:/ClientApplication.war:{http://example_path/}Service1
-acquireProviderPolicyMethod [httpGet ]
-httpGetProperties [httpGetTargetURI http://example_path]]')
```

3. Save your changes to the master configuration.

   To save your configuration changes, enter the following command:

```
AdminConfig.save()
```

Results

The web application client-side policy is calculated when it is required at run time, based either on the policy of the service provider, or on the client policy set and the policy of the service provider, depending on which option you selected. This calculated policy is known as the “effective policy” and is cached as a runtime configuration. The effective policy is used for subsequent outbound web service requests to the endpoint or operation for which the dynamic policy calculation was performed. The policy set configuration of the client does not change.

The provider policy that the client holds for a service is refreshed the first time that the web service is invoked after the application is loaded. After that, the provider policy is refreshed when the application restarts, or if the application explicitly invokes a refresh. When the provider policy is refreshed, the effective policy is recalculated.

Configuring the client policy to use a service provider policy from a registry:

An application that is a web service client can obtain the policy configuration of a web service provider from a registry, such as WebSphere Service Registry and Repository (WSRR), and use this information to establish a policy configuration that is acceptable to both the client and the service provider.

Before you begin

You have developed a web service client that contains all the necessary artifacts, and deployed your web services application into your application server instance. If you require them, you have attached the policy sets and managed the associated bindings.

The Web Services Description Language (WSDL) for the policy of the service provider, and its corresponding policies and policy attachments, are stored in a registry such as WSRR. That policy must contain its policy configuration in WS-PolicyAttachments format. The client must be able to support those provider policies.

The registry must support the use of HTTP GET requests to publish WSDL that contains WS-Policy attachments, for example WSRR Version 6.2 or later.

For a list of WS-Policy assertion specifications and WS-Policy domains that are supported, see the WS-Policy topic.

About this task

You can administer the client to configure itself dynamically at run time, based on the policy of a service provider that is held in a registry. You can administer the client at the service or service reference level to dynamically apply the provider policy that it obtains from a registry. By default, endpoints and operations inherit their policy configuration from the relevant service. However, it is possible to configure a service reference to override the service, in which case the endpoints and operations inherit their policy configuration from the service reference. You cannot administer the client to apply dynamically the provider policy that it obtains from a registry at the application level.

You can configure the client policy to use a service provider policy that is stored in a registry by using the administrative console. You can also configure the client policy to use a service provider policy that is stored in a registry by using wsadmin commands.

Procedure

1. From the navigation pane of the administrative console, click **Applications** > **Application Types** > **WebSphere enterprise applications**.
2. Click the web service client application that you want to configure.
3. Click **[Web services properties] Service client policy sets and bindings**.
4. In the row for the service where you want to apply the policy, click the link in the Policies Applied column. You cannot apply the policy at application level. The Policies Applied pane is displayed.
5. Select one of the following options from the drop-down list:
   - Provider policy only. Configure the client based solely on the policy of the service provider. This option is available when a client policy set is not attached.
   - Client and provider policy. Configure the client based on both the client policy set and the policy of the service provider. This option is available when a client policy set is attached.

   The other options in the list do not apply to this task.
6. Click **HTTP GET request**.
7. Click **Specify request target**, then enter the URL for the location of the provider policy in the field, that is, the address in the repository for the WSDL and policy. For information about using WSRR to

retrieve a WSDL document with embedded policies, and therefore obtain the required URL, see the WSRR documentation. The following example shows a typical URL:

```
https://www.wsrr.host/WSRR/6.2/PolicyService/
WSDL?bsrURI=3b9b493b-278f-4f64.ba3f.dabd30da3f7e
```

8. Click **OK**.
9. Optional: If there is a secure connection that uses the Secure Sockets Layer (SSL) protocol between the client and the registry, ensure that trust is established between the application server and the registry server. To access the registry, the client uses the SSL transport policy that is part of its service-level application policy. For example, for WSRR, you can enter the URL for the WSRR server in a browser window. If the WSRR server is not already trusted, a message is displayed stating that the security certificate is not trusted. To establish trust, use the following steps:
   a. Retrieve and store the X509 certificate from the WSRR server. Use the options on the message to view details of the certificate and save those details to a file, using distinguished encoding rules (DER) encoded binary format.
   b. Find out the key store that the client uses, that is, the key store that is shown by the SSL security transport bindings of the client application policy set. See Configuring the SSL transport policy. For example, the key store might be the default trust store for the node.
   c. Add the signer certificate that you saved in step a. to the key store that the client uses. See Adding a signer certificate to a keystore.
10. Optional: To access the registry, the client uses the transport policy that is part of its service-level application policy. If the registry requires authentication using the HTTP protocol, configure a valid user name and password as part of the application-level transport policy binding configuration. It is advisable to secure any authorization credentials, because they are used for interactions with both the web service endpoint and the registry.
    a. Ensure that the client has a policy set that contains the HTTP transport policy attached to the application or service level. See the relevant steps in Managing policy sets and bindings for service clients at the application level using the administrative console .
    b. Configure the HTTP transport client bindings for the binding named Client sample and enter the user name and password that the registry requires to authenticate outbound service requests. See the relevant steps in Configuring the HTTP transport policy.
11. Save your changes to the master configuration.

Results

The web application client-side policy is calculated when it is required at run time, based either on the policy of the service provider, or on the client policy set and the policy of the service provider, depending on which option you selected. This calculated policy is known as the “effective policy” and is cached as a runtime configuration. The effective policy is used for subsequent outbound web service requests to the endpoint or operation for which the dynamic policy calculation was performed. The policy set configuration of the client does not change.

The provider policy that the client holds for a service is refreshed the first time that the web service is invoked after the application is loaded. After that, the provider policy is refreshed when the application restarts, or if the application explicitly invokes a refresh. When the provider policy is refreshed, the effective policy is recalculated.

Policies applied settings:

Use this panel to view and change whether the policy configuration of a WebSphere Application Server service client is configured dynamically, based on the policies supported by its service provider. You can view or change how the client obtains the policy of the service provider; the client can use an HTTP GET request or a Web Services Metadata Exchange (WS-MetadataExchange) request. You can specify a policy set and binding to provide message-level security for WS-MetadataExchange requests or to specify HTTP transport and SSL transport configuration for HTTP GET requests.

To view this administrative console page for an application or service, complete the following steps. The application must be a web services service client.

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***service_client_application_name*** > **[Web Services Properties] Service client policy sets and bindings**.
2. Select the link in the **Policies Applied** column for the application or service you require.

To view this administrative console page for a service, complete the following steps:

1. Click **Services** > **Service clients** > ***service_name***.
2. Select the link in the **Policies Applied** column for the service. The link is available if the service or the parent application has a policy set attached.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Apply the following policies:

Specifies whether the client policy is based on the policy of the service provider, and how that policy is used.

Select from the following options:

- Client policy only. Configure the client based solely on the client policy set. Do not use the policy of the service provider. This option is available when a client policy set is attached to the resource.
- Client and provider policy. Configure the client based on both the client policy set and the policy of the service provider. This option is available when a client policy set is attached to the resource.
- Inherit application attachment. Inherit the setting of the parent application. This option is available for a service when a client policy set is not attached to the service. If there is a policy set attached to the parent application, the inherited properties are displayed on this panel, but you cannot change them. If there is no policy set attached to the parent application, when you return to the Service clients policy sets and bindings panel, the Policies applied column shows a value of **None**.
- Provider policy only. Configure the client based solely on the policy of the service provider. This option is available when a client policy set is not attached to the resource.

HTTP GET request:

Click **HTTP GET request** to obtain the policy of the service provider by using an HTTP GET request. The policy configuration must be in WS-PolicyAttachments format in the WSDL of the service provider.

This option is available when **Apply the following policies** is set to Client and provider policy or Provider policy only.

By default, the HTTP GET request is targeted at the URL for each service endpoint followed by ?WSDL.

Use the default request target:

When you apply a policy to a service, click **Use the default request target** to target the HTTP GET request at the URL for each service endpoint followed by ?WSDL.

If the attach point is for the service, then you can either select this default request target or you can choose to specify an alternative request target using the **Specify request target** option.

If the attach point is for the application then the default request target will be used.

Specify request target:

When you apply a policy to a service, click **Specify request target** to change the target for acquiring provider policy using an HTTP GET request. Enter the URL for the location of the provider policy in the field.

This option is available when **HTTP GET request** is selected and you apply a policy to a service.

When you apply a policy to an application, this option is not available.

Attach a system policy set to the HTTP GET request:

Select **Attach a system policy set to the HTTP GET request** to set HTTP transport and SSL transport configuration for the HTTP GET request. This option is available when **HTTP GET request** is selected as the method to be used to obtain the provider policy and when **Specify request target** is selected and completed.

If you do not specify a policy set you will inherit the HTTP transport and SSL transport configuration from the application.

Policy set (for the HTTP GET request):

Select the policy set you require from the list to provide HTTP transport and SSL transport configuration for the HTTP GET request. Select from system policy sets that contain solely HTTP transport policies, solely SSL transport policies, or both; the policy set cannot contain other policy types.

This option is available when **Attach a system policy set to the HTTP GET request** is selected and the **Specify request target** is selected and completed.

Binding (for the HTTP GET request):

Select the binding you require from the list for the HTTP GET request. You can select from **Global Default Bindings** or **General client/provider policy set bindings**, which are specific to the individual service.

This option is available when **Attach a system policy set to the HTTP GET request** is selected and the **Specify request target** is selected and completed.

The value of **Default** will result in the Global Default Binding being used.

WS-MetadataExchange request:

Click **WS-MetadataExchange** to obtain the policy of the service provider by using a WS-MetadataExchange GetMetadata request. The policy configuration must be in WS-PolicyAttachments format in the WSDL of the service provider.

This option is available when **Apply the following policies** is set to Client and provider policy or Provider policy only.

Attach a system policy set to the WS-MetadataExchange request:

Select **Attach a system policy set to the WS-MetadataExchange request** to set message-level security for the WS-MetadataExchange request. By default, this option is not selected and the transport policy of the application is used. This option is available when **WS-MetadataExchange request** is selected.

When **Attach a system policy set to the WS-MetadataExchange request** is selected, the **Policy set** and **Binding** lists are available. If you select **Attach a system policy set to the WS-MetadataExchange request**, you must also select a policy set and a binding.

Policy set (for the WS-MetadataExchange request):

Select the policy set you require from the list to provide message-level security for the WS-MetadataExchange request. You can select from system policy sets that contain only WS-Security policies, only WS-Addressing policies, or both. The default policy set is SystemWSSecurityDefault.

System policy sets are used for system messages that are not business-related, for example, messages that apply qualities of service (QoS), including the messages that are defined in the WS-MetadataExchange protocol.

Note that any transport policy of the application is always used.

This option is available when **Attach a system policy set to the WS-MetadataExchange** is selected.

Binding (for the WS-MetadataExchange request):

Select the binding you require from the list to provide message level security for the WS-MetadataExchange request. You can select from **Global Default Bindings** or **General client/provider policy set bindings**, which are specific to the individual service.

This option is available when **Attach a system policy set to the WS-MetadataExchange** is selected.

The value of **Default** will result in the Global Default Binding being used.

Enabling Web Services Addressing support for JAX-WS applications using deployment descriptors

For JAX-WS applications, you can enable WS-Addressing support during the packaging of either a service or client application, by editing the deployment descriptor for that application.

About this task

To modify WS-Addressing behavior by using deployment descriptors, add an `<addressing>` element to the deployment descriptor file for the application. The `<addressing>` element has optional child elements as described in the following table.

The `<addressing>` element functions in the same way as the Addressing annotation. The child elements of the `<addressing>` annotation function in the same way as the parameters of the Addressing annotation.

Table 142. Child elements of the addressing deployment descriptor element. The table lists the different child elements with their possible values and a description of each one.

Element name	Possible values	Description
`enabled`	`true` (default) `false`	Whether WS-Addressing support is enabled.
`required`	`true` `false` (default)	Whether WS-Addressing headers are required.

`responses`	`All` (default)
	`ANONYMOUS`
	`NON_ANONYMOUS`
		Whether to use a synchronous or an asynchronous message exchange pattern. Specify `ANONYMOUS` to send messages in a synchronous message pattern; use `NON_ANONYMOUS` to send messages in an asynchronous message exchange pattern.

Procedure

- To modify the behavior of the WS-Addressing support in the service application, add the <addressing> element, and optional child elements as required, to the service deployment descriptor under the <port-component> element within the <webservice-description> element. In the following example, the Addressing deployment descriptor fragment specifies that WS-Addressing is enabled and required, and that the asynchronous message exchange pattern is used.

```
<port-component>
     <port-component-name>MyPort1</port-component-name>
        <addressing>
         <enabled>true</enabled>
         <required>true</required>
         <responses>NON_ANONYMOUS</responses>
        </addressing>
        <service-impl-bean>
           <servlet-link>MyPort1ImplBean</servlet-link>
        </service-impl-bean>
</port-component>
```

- To modify the behavior of the WS-Addressing support in the client application, add the <addressing> element, and optional child elements as required, to the client deployment descriptor under the <port-component-ref> element within the <service-ref> element. For example, the following deployment descriptor fragment indicates that WS-Addressing is enabled:

```
<service-ref>
        <service-ref-name>service/MyPortComponentRef</service-ref-name>
        <service-interface>com.example.MyService</service-ref-interface>
        <port-component-ref>
           <service-endpoint-interface>com.example.MyPortType</service-endpoint-interface>
           <addressing>
              <enabled>true</enabled>
           </addressing>
        </port-component-ref>
</service-ref>
```

Results

WS-Addressing properties are now included in the SOAP message header, and are processed by the server on receipt of the message.

Enabling Web Services Addressing support for JAX-WS applications using addressing annotations

For JAX-WS applications, you can enable WS-Addressing support during development of a service application, by using addressing annotations in the code. You can also use this method in a client application that uses an injected web service proxy reference.

About this task

Use one of the following addressing annotations in your service code:

- Addressing - use this annotation if you want to use the 2005/08 WS-Addressing specification.
- SubmissionAddressing - use this annotation if you want to use the 2004/08 WS-Addressing specification.

On clients, use the Addressing annotation only; the SubmissionAddressing annotation is not supported. You must specify the Addressing annotation in combination with the WebServiceRef annotation. The WebServiceRef annotation specifies a reference to the web service proxy that is injected by the client container.

Annotation settings override settings in the WSDL document. Annotation settings might differ from WSDL settings if you create the WSDL document manually rather than generating it from the code.

Specify up to three optional parameters for each annotation:

Table 143. Parameters for the addressing annotations. The table lists the different parameters with their possible values and a description of each one.

Parameter name	Possible values	Description
`enabled`	`true` (default) `false`	Whether WS-Addressing support is enabled.
`required`	`true` `false` (default)	Whether WS-Addressing headers are required.

`responses`	`Responses.All` (default)
	`Responses.ANONYMOUS`
	`Responses.NON_ANONYMOUS`
		Whether to use a synchronous or an asynchronous message exchange pattern. Specify `Responses.ANONYMOUS` to send messages in a synchronous message pattern; use `Responses.NON_ANONYMOUS` to send messages in an asynchronous message exchange pattern. **Note:** This parameter is not supported on the SubmissionAddressing annotation.

Note: You can use the Addressing annotation only with a SOAP (1.1 or 1.2) over HTTP binding. If you use the class with another binding, such as XML over HTTP, an exception is thrown on clients, and on servers the web service fails to deploy.

Procedure

- To modify the behavior of the WS-Addressing support programmatically in the service application, use one of the addressing annotations, with optional parameters as required, in the code. In the following example, the Addressing annotation is used with no parameters, so the default settings apply.

```
import javax.xml.ws.soap.Addressing;

@Addressing
@WebService(endpointInterface =
   "org.apache.axis2.jaxws.calculator.Calculator",
   serviceName = "CalculatorService",
   portName = "CalculatorServicePort",
   targetNamespace = "http://calculator.jaxws.axis2.apache.org")
```

  In the following example, the SubmissionAddressing annotation is used with parameters that specify that WS-Addressing is enabled and required. The `responses` attribute is not supported on this annotation.

```
import com.ibm.websphere.wsaddressing.jaxws21.SubmissionAddressing;

@SubmissionAddressing(enabled=true, required=true)
@WebService(endpointInterface =
   "org.apache.axis2.jaxws.calculator.Calculator",
   serviceName = "CalculatorService",
   portName = "CalculatorServicePort",
   targetNamespace = "http://calculator.jaxws.axis2.apache.org")
```

- To enable WS-Addressing support on clients that use an injected web service proxy reference, use the Addressing annotation, with optional parameters as required, in combination with the WebServiceRef annotation. The SubmissionAddressing annotation is not supported for this method.

  For example, the following code fragment specifies that WS-Addressing is enabled and that the synchronous message exchange pattern is used:

```
public class MyClientApplication {

            // Enable Addressing for a port-component-ref resource injection.
            @Addressing(enabled=true, responses=Responses.ANONYMOUS)
            @WebServiceRef(MyService.class)
            private MyPortType myPort;
            ...
       }
```

Results

If you use an addressing annotation in the service application, the server processes any WS-Addressing headers that conform to the relevant specification in inbound SOAP messages. If you specify that WS-Addressing is required, and an inbound SOAP message does not include any WS-Addressing headers, or includes WS-Addressing headers that do not conform to the specification indicated by the

annotation, the server returns a fault message. For example, if a client sends a message that includes 2004/08 WS-Addressing headers, and the server requires 2005/08 headers, the server returns a fault message.

If you use the Addressing annotation and generate a WSDL document from the code, a UsingAddressing element and WS-Policy assertions are created in the WSDL document. Clients that use this WSDL document will include WS-Addressing information in their messages. The SubmissionAddressing annotation is not understood by current WSDL generation tools. However, the WSDL document does not distinguish between the 2005/08 specification and the 2004/08 specification, so if you want to generate a WSDL document from code that contains a SubmissionAddressing annotation, use both the Addressing and SubmissionAddressing annotations together.

If you specify the responses attribute, the corresponding message exchange pattern will be used.

Enabling Web Services Addressing support for JAX-WS applications using addressing features

For JAX-WS applications, you can enable WS-Addressing support during development of a client application, by using addressing features in the code.

About this task

Use one of the following addressing feature classes in your client code:

- AddressingFeature - use this class if you want to send messages that include WS-Addressing headers that conform to the 2005/08 WS-Addressing specification
- SubmissionAddressingFeature - use this class if you want to send messages that include WS-Addressing headers that conform to the 2004/08 WS-Addressing specification

If you use both feature classes, the specification that is used depends on the type of endpoint reference that you also specify. For example, if you specify a W3CEndpointReference object, the specification that is used is the 2005/08 specification. If you specify an endpoint reference whose type conflicts with that indicated by the feature class, for example a W3CEndpointReference object with a SubmissionAddressingFeature instance, an error is thrown. If you do not specify an endpoint reference, the default specification is the 2005/08 specification.

Specify up to three optional parameters for each addressing feature instance:

Table 144. Parameters for the addressing features. The table lists the different addressing feature parameters as well as their possible values and a description of each one.

Parameter name	Possible values	Description
`enabled`	`true` (default) `false`	Whether WS-Addressing support is enabled.
`required`	`true` `false` (default)	Whether WS-Addressing headers are required.

`responses`	`Responses.All` (default)
	`Responses.ANONYMOUS`
	`Responses.NON_ANONYMOUS`
		Whether to use a synchronous or an asynchronous message exchange pattern. Specify Responses.ANONYMOUS to send messages in a synchronous message pattern; use Responses.NON_ANONYMOUS to send messages in an asynchronous message exchange pattern.
		Note: This parameter is not supported for the SubmissionAddressingFeature class.

Note: You can use the addressing feature classes only with a SOAP (1.1 or 1.2) over HTTP binding. If you use the class with another binding, such as XML over HTTP, an exception is thrown on clients, and on servers the web service fails to deploy.

Procedure

Create an instance of one of the addressing feature classes, with parameters as required. For example, to specify that WS-Addressing is enabled and required, and that the 2005/08 specification and the asynchronous message exchange pattern is used, use the following code:

```
AddressingFeature feat = new AddressingFeature(true, true, AddressingFeature.Responses.NON_ANONYMOUS);
```

To specify that WS-Addressing is disabled for the 2004/08 specification, use the following code:

```
SubmissionAddressingFeature feat = new SubmissionAddressingFeature(false);
```

Results

If you specify that WS-Addressing is enabled, the client includes WS-Addressing headers in SOAP messages. The headers conform to the WS-Addressing specification indicated by the type of feature class used. If the server does not use annotations, or uses policy sets to enable WS-Addressing, the server accepts both the 2005/08 and 2004/08 specifications.

If you specify that WS-Addressing is required and the client receives a message that does not include WS-Addressing headers, the client returns a fault.

If you specify the responses attribute, the corresponding message exchange pattern will be used.

Enabling Web Services Addressing support for JAX-WS applications using WS-Policy

For JAX-WS applications, you can enable WS-Addressing support during the development of a client or service application by adding WS-Policy assertions into the WSDL document.

About this task

The JAX-WS 2.2 specification introduces functionality that enables WS-Policy assertions in the application WSDL document to be mapped to and from Java annotations in the code.

There are several ways of enabling WS-Addressing support for JAX-WS applications. The method of using WS-Policy will be effective unless it is overridden by another method with greater precedence, such as if you are using AddressingFeature classes or using Addressing annotations. See the parent topic: Enabling Web Services Addressing support for JAX-WS applications for full details about the order of precedence used by WebSphere Application Server.

Note: WebSphere Application Server version 8 supports the JAX-WS 2.2 specification. One of the differences between the JAX-WS 2.1 and JAX-WS 2.2 specifications is that whereas the presence of WS-Policy in an application's WSDL was formerly ignored, if WS-Addressing support is defined within the WS-Policy, this will now be used by WebSphere Application Server in the configuration of the application. As the presence of WS-Policy in an application's WSDL is now checked for WS-Addressing configuration, you may notice a change in behavior in applications formerly run in previous versions of WebSphere Application Server.

If you add an Addressing annotation to your provider code and use the **wsgen** command-line tool to generate the WSDL document, it will contain WS-Policy assertions specifying the WS-Addressing support. However, if you are looking at the published WSDL document for the provider service, it is possible that policy defined in policy sets is showing, as this would overwrite any annotations defined in the code.

If you are using an existing WSDL document to create Java code, when WebSphere Application Server reads the WSDL, if any WS-Policy assertions specifying WS-Addressing support are present, the generated Java code will contain Addressing annotations.

Procedure

- To enable WS-Addressing support on clients, use WS-Policy assertions in the code, with optional attributes as required. The `wsam:Addressing` assertion indicates that WS-Addressing is required. If you want to indicate that WS-Addressing is supported but not mandatory, add the `wsp:Optional` attribute. In the following example, WS-Addressing is supported, but is not mandatory and the messaging format to use has not been specified.

```
<wsp:Policy>
    <wsam:Addressing wsp:Optional="true">
        <wsp:Policy/>
    </wsam:Addressing>
</wsp:Policy>
```

  This example is equivalent to the policy set configuration of WS-Addressing being set to non-mandatory and the messaging style being set to synchronous and asynchronous.
- To specify the message exchange pattern to be employed, use the `wsam:AnonymousResponses` assertion for synchronous message exchanges and the `wsam:NonAnonymousResponses` assertion for asynchronous message exchanges. In the following example, WS-Addressing support is set as mandatory and a synchronous message exchange pattern has been specified.

```
<wsp:Policy>
    <wsam:Addressing>
        <wsp:Policy>
            <wsam:AnonymousResponses/>
        </wsp:Policy>
    </wsam:Addressing>
</wsp:Policy>
```

  This example is equivalent to the policy set configuration of WS-Addressing being set to required and the messaging style being set to synchronous only.

  For more information, see the Web Services Addressing 1.0 - Metadata specification document.

Results

If you add WS-Addressing annotations to the application code specifying that WS-Addressing is enabled, when you generate the WSDL it will contain WS-Policy assertions.

When WS-Policy assertions specifying WS-Addressing support are included in the WSDL, WS-Addressing headers are included in the generated SOAP messages.

If you specify that WS-Addressing is required and an inbound SOAP message is received that does not include any WS-Addressing headers, an exception occurs.

Web Services Addressing annotations

The WS-Addressing specification provides transport-neutral mechanisms to address web services and to facilitate end-to-end addressing. If you have a JAX-WS application you can use Java annotations in your code to specify WS-Addressing behavior at run time.

You can use WS-Addressing annotations to enable WS-Addressing support, to specify whether WS-Addressing information is required in incoming messages, to control the message exchange pattern the service supports, and to specify actions to be associated with a web service operation or fault response.

The following WS-Addressing annotations are supported in WebSphere Application Server. These annotations are defined in the JAX-WS 2.2 specification unless otherwise stated. The JAX-WS 2.2 specification supersedes and includes functions within the JAX-WS 2.1 specification. See the Java API for XML-Based Web Services 2.2 specification for full details.

javax.xml.ws.Action
: Specifies the action that is associated with a web service operation.

- When following a bottom-up approach to developing JAX-WS web services, you can generate a WSDL document from Java application code using the **wsgen** command-line tool. However, for this attribute to be added to the WSDL operation, you must also specify the @Addressing annotation on the implementation class. The result in the generated WSDL document is that the Action annotations will have the `wsam:Action` attribute on the `input message` and `output message` elements of the `wsdl:operation`. Alternatively, if you do not want to use the @Addressing annotation you can supply your own WSDL document with the Action attribute already defined.
- When following a top-down approach to developing JAX-WS web services, you can generate Java application code from an existing WSDL document using the **wsimport** command-line tool. In such cases, the resulting Java code will contain the correct Action and FaultAction annotations.

If this action is not specified in either code annotations or in the WSDL document, the default action pattern as defined in the Web Services Addressing 1.0 Metadata specification is used. Refer to this specification for full details.

Note: Whilst the WebSphere Application Server runtime environment supports the deprecated `wsaw:Action` attribute, if you try to generate Java code from an old WSDL document containing the deprecated `wsaw:Action` attribute, this attribute will be ignored.

javax.xml.ws.FaultAction

Specifies the action that is added to a fault response. When you use this annotation with a particular method, the WS-Addressing FaultAction extension attribute is added to the fault element of the WSDL operation that corresponds to that method. For this attribute to be added to the WSDL operation, you must also specify the Addressing annotation on the implementation class. If you do not want to use the Addressing annotation you can supply your own WSDL document with the Action attribute already defined. This annotation must be contained within an Action annotation.

WSDL documents generated from Java application code containing the WS-Addressing FaultAction annotation will have the `wsam:Action` attribute on the `fault message` element of the `wsdl:operation`.

Note: To ensure that any custom Exception classes you write are successfully mapped to the generated WSDL document, extend the java.lang.Exception class instead of the java.lang.RuntimeException class.

javax.xml.ws.soap.Addressing

Specifies that this service is to enable WS-Addressing support. You can use this annotation only on the service implementation bean; you cannot use it on the service endpoint interface.

com.ibm.websphere.wsaddressing.jaxws21.SubmissionAddressing

This annotation is part of the IBM implementation of the JAX-WS specification. This annotation specifies that this service is to enable WS-Addressing support for the 2004/08 WS-Addressing specification. You can use this annotation only on the service implementation bean; you cannot use it on the service endpoint interface.

For more information about the Addressing and SubmissionAddressing annotations, including code examples, see Enabling Web Services Addressing support for JAX-WS applications using addressing annotations.

The following example code uses the Action annotation to define the invoke operation to be invoked (input), and the action that is added to the response message (output). The example also uses the FaultAction annotation to specify the action that is added to a response message if a fault occurs:

```
@WebService(name = "Calculator")
public interface Calculator {
   ...
    @Action(
      input="http://calculator.com/inputAction",
      output="http://calculator.com/outputAction",
```

```
      fault = { @FaultAction(className=AddNumbersException.class,
                value="http://calculator.com/faultAction")
      }
   )
   public int add(int value1, int value2) throws AddNumbersException {
      return value1 + value2;
   }
}
```

If you use a tool to generate service artifacts from code, the WSDL tags that are generated from the preceding example are as follows:

```
<definitions targetNamespace="http://example.com/numbers" ...>
   ...
   <portType name="AddPortType">
      <operation name="Add">
         <input message="tns:AddInput" name="Parameters"
                        wsam:Action="http://calculator.com/inputAction"/>
         <output message="tns:AddOutput" name="Result"
                         wsam:Action="http://calculator.com/outputAction"/>
         <fault message="tns:AddNumbersException" name="AddNumbersException"
                         wsam:Action="http://calculator.com/faultAction"/>
      </operation>
    <portType>
     ...
<definitions>
```

Web Services Addressing security

It is essential that communications that use Web Services Addressing (WS-Addressing) are adequately secured and that a sufficient level of trust is established between the communicating parties. You can achieve secure communications through the signing of WS-Addressing message-addressing properties and the encryption of endpoint references.

Undertake these actions for both the supported addressing namespaces, http://www.w3.org/2005/08/addressing and http://schemas.xmlsoap.org/ws/2004/08/addressing, even if you intend to use only one of those namespaces.

Signing of WS-Addressing message-addressing properties

You can use an assembly tool to specify the message-addressing properties, and therefore the WS-Addressing message elements, that require signing, or that require signature verification on inbound requests. The receiver of the message might rely on the presence of this verifiable signature to determine that the outbound message originated from a trusted source. Similarly, the lack of a verifiable signature that is associated with the specified inbound message addressing properties causes the rejection of the message with a SOAP fault.

Encryption of endpoint references

You can encrypt endpoint references as part of the SOAP header or SOAP body. Alternatively, you can remove the need for encryption by not including sensitive information in the address or reference parameters properties of the endpoint reference.

Use of the synchronous message exchange pattern

This method applies to JAX-WS applications only.

If you do not secure the WS-Addressing information in the SOAP message by using one or more of the previous methods, and you do not have WS-Security enabled, the ReplyTo and FaultTo elements of the SOAP message could be used to send messages to a third party, potentially taking part in a Denial of Service attack. To prevent such attacks, apply a WS-Addressing policy type and configure it to specify synchronous messaging only. You should also enable WS-Policy so that this requirement is communicated to clients.

Invoking JAX-WS web services asynchronously

Java API for XML-Based Web Services (JAX-WS) provides support for invoking web services using an asynchronous client invocation. JAX-WS provides support for both a callback and polling model when calling web services asynchronously. Both the callback model and the polling model are available on the Dispatch client and the Dynamic Proxy client.

Before you begin

Develop a JAX-WS Dynamic Proxy or Dispatch client. When developing Dynamic Proxy clients, after you generate the portable client artifacts from a Web Services Description Language (WSDL) file using the **wsimport** command, the generated service endpoint interface (SEI) does not have asynchronous methods included in the interface. Use JAX-WS bindings to add the asynchronous callback or polling methods on the interface for the Dynamic Proxy client. To enable asynchronous mappings, you can add the `jaxws:enableAsyncMapping` binding declaration to the WSDL file. For more information on adding binding customizations to generate an asynchronous interface, see chapter 8 of the JAX-WS specification.

Note: When you run the **wsimport** tool and enable asynchronous invocation through the use of the JAX-WS enableAsyncMapping binding declaration, ensure that the corresponding response message your WSDL file does not contain parts. When a response message does not contain parts, the request acts as a two-way request, but the actual response that is sent back is empty. The **wsimport** tool does not correctly handle a void response. To avoid this scenario, you can remove the output message from the operation which makes your operation a one-way operation or you can add a `<wsdl:part>` to your message. For more information on the usage, syntax and parameters for the **wsimport** tool, see the wsimport command for JAX-WS applications documentation.

About this task

An asynchronous invocation of a web service sends a request to the service endpoint and then immediately returns control to the client program without waiting for the response to return from the service. JAX-WS asynchronous web service clients consume web services using either the callback approach or the polling approach. Using a polling model, a client can issue a request and receive a response object that is polled to determine if the server has responded. When the server responds, the actual response is retrieved. Using the callback model, the client provides a callback handler to accept and process the inbound response object. The `handleResponse()` method of the handler is called when the result is available. Both the polling and callback models enable the client to focus on continuing to process work without waiting for a response to return, while providing for a more dynamic and efficient model to invoke web services. Polling invocations are valid from Enterprise JavaBeans (EJB) clients or Java Platform, Enterprise Edition (Java EE) application clients. Callback invocations are valid only from Java EE application clients.

Using the callback asynchronous invocation model

To implement an asynchronous invocation that uses the callback model, the client provides an `AsyncHandler` callback handler to accept and process the inbound response object. The client callback handler implements the `javax.xml.ws.AsyncHandler` interface, which contains the application code that is run when an asynchronous response is received from the server. The `javax.xml.ws.AsyncHandler` interface contains the `handleResponse(java.xml.ws.Response)` method that is called after the run time has received and processed the asynchronous response from the server. The response is delivered to the callback handler in the form of a `javax.xml.ws.Response` object. The response object returns the response content when the `get()` method is called. Additionally, if an error was received, then an exception is returned to the client during that call. The response method is then invoked according to the threading model used by the executor method, `java.util.concurrent.Executor` on the client's `java.xml.ws.Service` instance that was used to create the Dynamic Proxy or Dispatch client instance. The executor is used to invoke any asynchronous callbacks registered by the application. Use the `setExecutor` and `getExecutor` methods to modify and retrieve the executor configured for your service.

Using the polling asynchronous invocation model

Using the polling model, a client can issue a request and receive a response object that can subsequently be polled to determine if the server has responded. When the server responds, the actual response can then be retrieved. The response object returns the response content when the `get()` method is called. The client receives an object of type `javax.xml.ws.Response` from the `invokeAsync` method. That Response object is used to monitor the status of the request to the server, determine when the operation has completed, and to retrieve the response results.

Using an asynchronous message exchange

By default, asynchronous client invocations do not have asynchronous behavior of the message exchange pattern on the wire. The programming model is asynchronous; however, the exchange of request or response messages with the server is not asynchronous. To use an asynchronous message exchange, the `com.ibm.websphere.webservices.use.async.mep` property must be set on the client request context with a boolean value of `true`. When this property is enabled, the messages exchanged between the client and server are different from messages exchanged synchronously. With an asynchronous exchange, the request and response messages have WS-Addressing headers added that provide additional routing information for the messages. Another major difference between asynchronous and synchronous message exchange is that the response is delivered to an asynchronous listener that then delivers that response back to the client. For asynchronous exchanges, there is no timeout that is sent to notify the client to stop listening for a response. To force the client to stop waiting for a response, issue a `Response.cancel()` method on the object returned from a polling invocation or a `Future.cancel()` method on the object returned from a callback invocation. The cancel response does not affect the server when processing a request.

Procedure

1. Determine if you want to implement the callback method or the polling method for the client to asynchronously invoke the web service.
2. (Optional) Configure the client request context. Add the

```
com.ibm.websphere.webservices.use.async.mep
```

property to the request context to enable asynchronous messaging for the web services client. Using this property requires that the service endpoint supports WS-Addressing which is supported by default for the application server. The following example demonstrates how to set this property:

```
Map<String, Object> rc = ((BindingProvider) port).getRequestContext();
 rc.put("com.ibm.websphere.webservices.use.async.mep", Boolean.TRUE);
```

3. To implement the asynchronous callback method, perform the following steps.
   a. Find the asynchronous callback method on the SEI or `javax.xml.ws.Dispatch` interface. For an SEI, the method name ends in *Async* and has one more parameter than the synchronous method of type `javax.xml.ws.AsyncHandler`. The `invokeAsync(Object, AsyncHandler)` method is the one that is used on the Dispatch interface.
   b. (Optional) Add the `service.setExecutor` methods to the client application. Adding the executor methods gives the client control of the scheduling methods for processing the response. You can also choose to use the `java.current.Executors` class factory to obtain packaged executors or implement your own executor class. See the JAX-WS specification for more information on using executor class methods with your client.
   c. Implement the `javax.xml.ws.AsyncHandler` interface. The `javax.xml.ws.AsyncHandler` interface only has the `handleResponse(javax.xml.ws.Response)` method. The method must contain the logic for processing the response or possibly an exception. The method is called after the client run time has received and processed the asynchronous response from the server.
   d. Invoke the asynchronous callback method with the parameter data and the callback handler.
   e. The `handleResponse(Response)` method is invoked on the callback object when the response is available. The `Response.get()` method is called within this method to deliver the response.
4. To implement the polling method,

a. Find the asynchronous polling method on the SEI or `javax.xml.ws.Dispatch` interface. For an SEI, the method name ends in *Async* and has a return type of `javax.xml.ws.Response`. The `invokeAsync(Object)` method is used on the Dispatch interface.

b. Invoke the asynchronous polling method with the parameter data.

c. The client receives the object type, `javax.xml.ws.Response`, that is used to monitor the status of the request to the server. The `isDone()` method indicates whether the invocation has completed. When the `isDone()` method returns a value of `true`, call the `get()` method to retrieve the response object.

5. Use the `cancel()` method for the callback or polling method if the client needs to stop waiting for a response from the service. If the `cancel()` method is invoked by the client, the endpoint continues to process the request. However, the wait and response processing for the client is stopped.

Results

You have enabled your JAX-WS web service client to asynchronously invoke and consume web services. See the JAX-WS specification for additional information regarding the asynchronous client APIs.

Example

The following example illustrates a web service interface with methods for asynchronous requests from the client.

```
@WebService

public interface CreditRatingService {
       // Synchronous operation.
       Score getCreditScore(Customer     customer);
       // Asynchronous operation with polling.
       Response<Score> getCreditScoreAsync(Customer customer);
       // Asynchronous operation with callback.
       Future<?> getQuoteAsync(Customer customer,
              AsyncHandler<Score> handler);
}
```

Using the callback method

The callback method requires a callback handler that is shown in the following example. When using the callback procedure, after a request is made, the callback handler is responsible for handling the response. The response value is a response or possibly an exception. The `Future<?>` method represents the result of an asynchronous computation and is checked to see if the computation is complete. When you want the application to find out if the request is completed, invoke the `Future.isDone()` method. Note that the `Future.get()` method does not provide a meaningful response and is not similar to the `Response.get()` method.

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerTom,
       new AsyncHandler<Score>() {
              public void handleResponse (
                     Response<Score> response)
                  {
                     score = response.get();
                     // process the request...
                  }
    }
);
```

Using the polling method

The following example illustrates an asynchronous polling client:

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerTom);

while (!response.isDone()) {
       // Do something while we wait.
}

score = response.get();
```

Enabling Web Services Addressing support for JAX-RPC applications

The Web Services Addressing (WS-Addressing) support provides mechanisms to address web services and provide addressing information in messages. To enable the WS-Addressing support for JAX-RPC applications, either configure the Web Services Description Language (WSDL) file for a service that runs on WebSphere Application Server, or use the WS-Addressing application programming interface (API) or system programming interface (SPI) to add WS-Addressing properties in a WebSphere Application Server client.

About this task

Complete this task to enable the WS-Addressing support, either as a service provider or as a client of a service provided by another party. This task also describes how to disable the WS-Addressing support, which can improve performance for those applications that do not use WS-Addressing or any protocol that depends on the WS-Addressing support.

If you are creating a web service, you can enable the WS-Addressing support during development of the service, by including the UsingAddressing extensibility element in the WSDL binding element for the service. This element contains a **required** attribute that has a value of either `false`, which specifies that WS-Addressing information is accepted but not required in incoming messages, or `true`, which specifies that WS-Addressing information is required in incoming messages. The default value is `false`. Messages from WebSphere Application Server clients always include WS-Addressing information if your service WSDL file includes the UsingAddressing element, regardless of the value of the **required** attribute.

If you are creating a client application to use a service from another provider, you might not have access to the WSDL file for the service, or the service might use a version of WSDL that does not support the UsingAddressing element (if the service is not running on a current version of this product). However, you can still enable WS-Addressing support, during run time, by setting WS-Addressing properties on the JAX-RPC stub or call object that you use to communicate with the service.

The following table summarizes the behavior of the WS-Addressing support in each of the scenarios mentioned previously.

Table 145. The behavior of the WS-Addressing support in the product. The table details the resulting behavior of different `UsingAddressing` settings when an application server client sends a message.

	The WSDL for the service specifies `UsingAddressing required = "false"`	**The WSDL for the service specifies**	**The WSDL for the service does not specify UsingAddressing**
A client sends a message that contains WS-Addressing information	The WS-Addressing information is processed by the product.	The WS-Addressing information is processed by the product.	The WS-Addressing information is processed by the product.
A non-WebSphere Application Server client sends a message that does not contain WS-Addressing information	The message is accepted.	The service returns a fault.	The message is accepted.
A WebSphere Application Server client sends a message, without specifying addressing properties	The message automatically contains the mandatory WS-Addressing information, as defined in the WS-Addressing specification. The information is processed by the product.	The message automatically contains the mandatory WS-Addressing information, as defined in the WS-Addressing specification. The information is processed by the product.	WS-Addressing information is not added. The message is accepted.

Procedure

- To enable WS-Addressing support from the server by configuring the WSDL file, complete the following steps:
  1. Ensure that the WSDL file for the service contains the UsingAddressing extensibility element on the binding element. If you generated the WSDL file by using the Java2WSDL tool, this element is

automatically added for you. If you created the WSDL file yourself, for use with the WSDL2Java tool, you must add the extensibility element. The UsingAddressing element has a **required** attribute with a default value of `false`. For example:

```
<wsdl:binding name="TestServiceSoapBinding" type="intf:TestService">
     <wsaw:UsingAddressing wsdl:required="false"
                           xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"/>

     <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

     <wsdl:operation name="invokeInstance">
       ...
     </wsdl:operation>

  </wsdl:binding>
```

This code indicates that the endpoint will process WS-Addressing information, but that this information is not required.

2. Optional: To specify that WS-Addressing information is required, change the value of the **required** attribute to `true`. If the endpoint receives a message that does not contain the mandatory WS-Addressing elements within the message header, the endpoint returns a fault message, as defined in the WS-Addressing specification.

   WebSphere Application Server clients and Proxy Server for IBM WebSphere Application Server always send WS-Addressing conformant messages to endpoints with bindings that specify the UsingAddressing element.

- To enable WS-Addressing support from a WebSphere Application Server client, use the IBM proprietary WS-Addressing API or SPI to associate one or more WS-Addressing properties with the JAX-RPC stub or call object that is used to send messages to the endpoint.

  These properties become message-addressing properties (MAPs) in the SOAP message header. If the node that receives the message is a WebSphere Application Server node, it processes the incoming MAPs in accordance with the WS-Addressing specification, even if the service does not have a UsingAddressing element in its WSDL file.

  Use this method when communicating with endpoints that use earlier versions of the WS-Addressing specification (for example: http://schemas.xmlsoap.org/ws/2004/08/addressing) that do not support the UsingAddressing element, or when the WSDL file for the target endpoint is not available to the client.

Results

WS-Addressing properties are now included in the SOAP message header, and are processed by the server on receipt of the message.

Disabling Web Services Addressing support

The Web Services Addressing (WS-Addressing) support provides mechanisms to address web services and provide addressing information in messages. WS-Addressing support is disabled by default on clients. The method for disabling WS-Addressing support on servers depends on whether your application is based on JAX-RPC or JAX-WS.

About this task

You do not have to disable WS-Addressing support even if your application does not require it, because in most cases WS-Addressing support does not have a negative impact on the running of applications. For JAX-RPC applications, disabling WS-Addressing support can be risky as this action also disables support for other specifications such as Web Services Atomic Transactions.

Procedure

- Disable WS-Addressing support for JAX-WS service providers using one of the following ways:
  - Use both the Addressing and SubmissionAddressing annotations in the service code, with the **enabled** parameter set to `false`; for example:

```
import javax.xml.ws.soap.Addressing;

@Addressing(enabled=false)
@SubmissionAddressing(enabled=false)
@WebService(...)
```

  - Use the `<webservice-description>/<port-component>/<addressing>` deployment descriptor element in the deployment descriptor for the service application; for example:

```
<port-component>
   <port-component-name>MyPort1</port-component-name>
   <addressing>
      <enabled>false</enabled>
   </addressing>
   <service-impl-bean>
      <servlet-link>MyPort1ImplBean</servlet-link>
   </service-impl-bean>
</port-component>
```

- You do not have to take any action to disable WS-Addressing support for JAX-WS clients, because WS-Addressing support is disabled by default. However, you can programmatically specify that WS-Addressing is disabled by using one of the following ways:
  - Use both the AddressingFeature and SubmissionAddressingFeature classes in the client code, with the **enabled** parameter set to `false`; for example:

```
AddressingFeature feat = new AddressingFeature(false);
SubmissionAddressingFeature feat = new AddressingFeature(false);
```

  - Use the Addressing annotation for an injected web services proxy reference; for example:

```
public class MyClientApplication {
   // Disable Addressing for a port-component-ref resource injection.
   @Addressing(enabled=false)
   @WebServiceRef(MyService.class)
   private MyPortType myPort;
   ...
 }
```

  - Use the `<service>/<port-component>/<addressing>` deployment descriptor; for example:

```
<service-ref>
   <service-ref-name>service/MyPortComponentRef</service-ref-name>
   <service-interface>com.example.MyService</service-ref-interface>
   <port-component-ref>
      <service-endpoint-interface>com.example.MyPortType</service-endpoint-interface>
      <addressing>
         <enabled>false</enabled>
      </addressing>
   </port-component-ref>
</service-ref>
```

- To disable WS-Addressing support for JAX-RPC service providers or clients, set the **com.ibm.ws.wsaddressingAndDependentsDisabled** system property to `true`. For example:

```
java -Dcom.ibm.ws.wsaddressingAndDependentsDisabled=true ... application_name
```

  Attention: Use this property with care because applications might require WS-Addressing message addressing properties to function correctly. Setting this property also disables support for the following specifications, which depend on the WS-Addressing support: Web Services Atomic Transactions, Web Services Business Agreement, Web Services Notification and Web Services Reliable Messaging.

Results

By completing this task, you disabled the WS-Addressing support. Disabling WS-Addressing on clients prevents WebSphere Application Server sending WS-Addressing message addressing properties in the SOAP header of outbound web service messages. Disabling WS-Addressing on servers additionally prevents WebSphere Application Server processing WS-Addressing MAPs in incoming SOAP headers.

Chapter 30. Developing web services - Invocation framework (WSIF)

The Web Services Invocation Framework (WSIF) is a Web Services Description Language (WSDL)-oriented Java™ API. You use this API to invoke web services dynamically, regardless of the service implementation format (for example enterprise bean) or the service access mechanism (for example Java Message Service (JMS)). Using WSIF, you can move away from the usual web services programming model of working directly with the SOAP APIs, towards a model where you interact with representations of the services. You can therefore work with the same programming model regardless of how the service is implemented and accessed.

Using WSIF to invoke web services

You invoke a web service dynamically by using the WSIF API directly.

About this task

You specify the location of the Web Services Description Language (WSDL) file for the service, the name of the operation to invoke, and any operation arguments. All other information needed to access the web service (the abstract interface, the binding, and the service endpoint) is available through the WSDL.

This kind of invocation does not require stub classes and does not need a separate compilation cycle.

Note: You should not use WSIF for new applications in WebSphere Application Server, unless you are supporting an existing WSIF configuration. You should instead adopt a more recent open standard, such as the Java API for XML-Based Web Services (JAX-WS) programming model.

For more information about using the Web Services Invocation Framework (WSIF) to invoke web services, see the following topics:

Procedure

- “Linking a WSIF service to the underlying implementation of the service.”
- “Developing a WSIF service” on page 1385.
- “Using complex types” on page 1393.
- “Using WSIF to bind a JNDI reference to a web service” on page 1394.
- “Example: Passing SOAP messages with attachments by using WSIF” on page 1396.
- “Interacting with the Java EE container in WebSphere Application Server” on page 1398.
- “Invoking a WSDL-based web service through the WSIF API” on page 1399.
- “Running WSIF as a client” on page 1405

Linking a WSIF service to the underlying implementation of the service

A Web Services Invocation Framework (WSIF) service is linked to the underlying service through a WSIF provider. A provider is an implementation of a Web Services Description Language (WSDL) binding that can run a WSDL operation through a binding-specific protocol. Providers implement the interface between the WSIF API and the implementation of a service.

About this task

Providers are pluggable within the WSIF framework, and are registered according to the namespace of the WSDL extension that they implement. Some providers use the Java Platform, Enterprise Edition (Java EE) programming model to use Java EE services. If a provider is available, but its required class libraries are not, then the provider is disabled.

To use the providers that are supplied with WebSphere Application Server, see the following topics:

Procedure

- Link a WSIF service to a SOAP over HTTP service.
- Link a WSIF service to a JMS-provided service (SOAP over JMS, or native JMS).
- Link a WSIF service to a local Java application.
- Write the WSDL extension that lets your WSIF service invoke an enterprise bean.

Linking a WSIF service to a SOAP over HTTP service

The SOAP provider allows WSIF stubs and dynamic clients to invoke SOAP services. Add Web Services Description Language (WSDL) extensions to your web service WSDL file so that the service can use the SOAP provider.

Before you begin

Note: The current WSIF default SOAP provider (the IBM Web Service SOAP provider) does not fully interoperate with services that are running on the former (Apache SOAP) provider. This is because the IBM Web Service SOAP provider is designed to interoperate fully with a JAX-RPC compliant web service, and Apache SOAP cannot provide such a service. For more information see "WSIF SOAP provider: working with existing applications."

About this task

The Web Services Invocation Framework (WSIF) SOAP provider supports SOAP 1.1 over HTTP.

The SOAP provider is JSR 101/109 compliant and uses Web Services for Java EE for parsing and creating SOAP messages.

The SOAP provider supports:
- SOAP-ENC encoding.
- RPC style and Document style SOAP messages.
- SOAP messages with attachments.

The SOAP provider is not transactional.

The SOAP provider does not support the WSIF synchronous timeout. The SOAP provider uses the default client timeout value that is set for Web Services for Java EE.

To link a WSIF service to a SOAP over HTTP service, you write extensions to the service WSDL file.

Procedure

- Optional: If you have a web service that you expect multiple clients to use to connect over SOAP, before you deploy the service set up your application deployment descriptor file `dds.xml` to handle multiple connections. For more information, see the troubleshooting tip Using WSIF with multiple clients causes a SOAP parsing error.
- Write the WSDL extension that lets your WSIF service access a SOAP over JMS service.

  Note: The WSDL binding extension for SOAP over JMS varies only slightly from the SOAP over HTTP binding.
- Write the WSDL extensions for SOAP attachments.

WSIF SOAP provider: working with existing applications:

The current WSIF default SOAP provider (the IBM Web Service SOAP provider) does not fully interoperate with services that are designed to run on the former (Apache SOAP) provider. This is because the IBM Web Service SOAP provider is designed to interoperate fully with a JAX-RPC compliant web service, and Apache SOAP cannot provide such a service.

About this task

As a result of the change from the Apache SOAP provider to the IBM Web Service SOAP provider, previous WSIF clients might not work in either of the following cases:

1. The web service uses any of the following parameter types: `xsd:date`, `xsd:dateTime`, `xsd:hexBinary` or `xsd:QName`. For more information, see the **Type Mappings** section of WSIF - Known restrictions.
2. The web service was built upon the Apache SOAP provider.

To get your existing services working again, you have two options:

Procedure

- Change the default WSIF SOAP provider back to the former Apache SOAP provider (in which case any future invocations to a JAX-RPC compliant web service will not work if that web service uses parameter types `xsd:date`, `xsd:dateTime`, `xsd:hexBinary` or `xsd:QName`).
- Modify your web services to use the IBM Web Service SOAP provider.

Changing the default WSIF SOAP provider:

The WSIF default SOAP provider (the IBM Web Service SOAP provider) is designed to interoperate fully with a JAX-RPC compliant web service, and therefore the default provider does not fully interoperate with services that are running on the former (Apache SOAP) provider. To get your existing services working again, you can either modify your web services to use the current IBM Web Service SOAP provider, or you can change the WSIF default provider back to Apache SOAP as described in this topic.

About this task

WSIF uses a properties file named `wsif.properties` to choose what SOAP provider to use. The SOAP provider is a node-wide setting, so all servers on the node must be restarted for any changes to take effect. The `wsif.properties` file is shipped in the `com.ibm.ws.runtime.jar` file that is located in the *app_server_root*`/plugins` directory (where *app_server_root* is the root directory for your installation of IBM WebSphere Application Server), and the “as shipped” properties file is accessed in this location by being put on the class path. However when you make changes to the file, you do not replace the original copy in the `com.ibm.ws.runtime.jar` file. Instead, you save the modified version in the *app_server_root*`/lib/properties` directory.

To change the WSIF default SOAP provider back to Apache SOAP, complete the following steps:

Procedure

1. Extract the `wsif.properties` file from the `com.ibm.ws.runtime.jar` file that is located in the *app_server_root*`/plugins` directory (where *app_server_root* is the root directory for your installation of IBM WebSphere Application Server).
2. Open the `wsif.properties` file in a text editor.
3. Remove the leading “#” character from the following lines:

```
# wsif.provider.default.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=1
# wsif.provider.uri.1.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=\
# http://schemas.xmlsoap.org/wsdl/soap/
#
```

After the update, the preceding lines should look like this:

```
wsif.provider.default.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=1
wsif.provider.uri.1.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=\
http://schemas.xmlsoap.org/wsdl/soap/
#
```

4. Save the updated `wsif.properties` file in the *app_server_root*`/lib/properties` directory.
5. Stop then restart all application servers on the node.

Example

Modifying web services to use the IBM Web Service SOAP provider:

The current WSIF default SOAP provider (the IBM Web Service SOAP provider) is designed to interoperate fully with a JAX-RPC compliant web service, and therefore the current default provider does not fully interoperate with services that are running on the former (Apache SOAP) provider. To get your existing services working again, you can either modify your web services to use the current IBM Web Service SOAP provider as described in this topic, or you can change the WSIF default provider back to Apache SOAP.

About this task

To modify an existing web service, use the assembly tool to complete the following steps and thereby generate new deployment artifacts for access to the service from the IBM Web Service provider:

Procedure

1. Import into the Workspace the project that contains your existing web services.
2. For every existing SOAP service in the project, repeat the following steps :
   a. From the pop-up menu for *your_service*`.wsdl`, select **Generate Deploy Code**.
   b. In the Generate Deploy Code window, change the **Inbound Binding Type** from `SOAP` to `IBM Web Service`.
   c. Click **Finish**.
3. Export the EAR file that contains all of the deployment artifacts for the IBM Web Service web service.

Linking a WSIF service to a JMS-provided service

The JMS providers enable a WSIF service to be invoked through either SOAP over JMS, or native JMS. Add Web Services Description Language (WSDL) extensions to your web service WSDL file so that the service can use the JMS providers.

About this task

The Java Message Service (JMS) is an API for transport technology. The mapping to a JMS destination is defined during deployment and maintained by the container.

The JMS destination endpoint for a web service can be realized in any of the following ways:
- The JMS destination for the queue can be the web service implementation.
- The JMS destination can be (but is not required to be) associated with a message-driven bean by the EJB container, thereby allowing the message-driven bean to be the web service implementation.
- For SOAP over JMS, the JMS destination can unwrap the JMS message and route the SOAP message to a web service that is implemented as a stateless session bean.

The JMS destination endpoint must respect the interaction model expected by the client and defined by the WSDL. It must return a response if one is required.

When the JMS destination endpoint creates the JMS response message the following rules must be followed:
- The response message must be sent to `JMSReplyTo` from the incoming request.

- The JMSCorrelationID value of the response message must be set to the JMSMessageID value from the request message.
- The response must be sent with a deliveryMode value equal to the JMSDeliveryMode value of the request message.
- The response must be sent with a priority value equal to the JMSPriority value of the request message.
- The TimeToLive/JMSExpiration value must be set to a value that equals the JMSExpiration value of the request message.

The client does not see any of these headers. The container receives the JMS message and (for SOAP over JMS) removes the SOAP message to send to the client.

To link a WSIF service to a JMS-provided service, use the following information and code examples:

Procedure

- Link your WSIF service to a SOAP over JMS service.
- Link a WSIF service to a service provided at a JMS destination.
- Enable a WSIF client to invoke a web service through JMS.

Writing the WSDL extension that lets your WSIF service access a SOAP over JMS service:

You can write a Web Services Description Language (WSDL) extension that enables your Web Services Invocation Framework (WSIF) service to access a SOAP service that uses the Java Message Service (JMS) as its transport mechanism.

Before you begin

This topic assumes that you chose and configured a JMS provider when you installed WebSphere Application Server (either the default messaging provider, or another provider such as the WebSphere MQ messaging provider). If not, do so now as described in Choosing a messaging provider.

About this task

If a SOAP message contains only XML, it can be carried on the Java Message Service (JMS) transport mechanism with the JMS message body type **TextMessage**. The SOAP message, including the SOAP envelope, is wrapped with a JMS message and put on the appropriate queue. The container receives the JMS message and removes the SOAP message to send to the client.

Use the following procedure, and associated code fragments, to help you to write the Web Services Description Language (WSDL) extension that enables your WSIF service to access a SOAP over JMS service.

Note: You can also use this procedure as a guide to writing the WSDL binding extension for SOAP over HTTP, because the SOAP over JMS binding is almost identical to the SOAP over HTTP binding.

Procedure

- Select the SOAP over JMS binding.

  You set the transport attribute of the <soap:binding> tag to indicate that JMS is used. If you also set the style attribute to rpc (Remote Procedure Call), then the Web Services Invocation Framework (WSIF) assumes that an operation is invoked on the web service endpoint:

  ```
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/jms"/>
  ```

- Set the JMS address.

  Note: See also the alternative method for specifying the JMS address that is given in the next step.

For SOAP over JMS, the <wsdl:port> tag must contain a <jms:address> element. This element provides the information required for a client to connect correctly to the web service by using the JMS programming model. Typically, it is the stubs generated to support the SOAP over JMS binding that act as the JMS client. Alternatively, the web service client can use the JMS programming model directly.

The <jms:address> element takes this form:

```
<jms:address

    destinationStyle="queue"
    jmsVendorURI="http://ibm.com/ns/mqseries"?
    initialContextFactory="com.ibm.NamingFactory"?
    jndiProviderURL="iiop://something:900/wherever"?
    jndiConnectionFactoryName="orange"
    jndiDestinationName="fred">

    <jms:propertyValue name="targetService" type="xsd:string"
                       value="StockQuoteServicePort"/>

</jms:address>
```

where attributes marked with a question mark (?) are optional.

The optional jmsVendorURI attribute is a string that uniquely identifies the JMS implementation. WSIF ignores this URI, which is used by the client developer and perhaps the client implementation to determine if it has access to the correct JMS provider in the client runtime environment.

The optional attributes initialContextFactory and jndiProviderURL can only be omitted if the runtime environment has a default Java Naming and Directory Interface (JNDI) provider configured.

The jndiConnectionFactoryName attribute gives the name of a JMS ConnectionFactory object, which can be looked up within the JNDI context given by the jndiContext attribute. This ConnectionFactory object is used to create a JMS connection to the JMS provider instance that owns the queue. In a simple configuration, the same ConnectionFactory object is used by the server message listener and by the clients. However the server and the clients can use different ConnectionFactory objects, provided that they all create connections to the same JMS provider instance.

The value attribute of the targetService <jms:propertyValue> element is the name of the port component for the target service as defined in the <port-component-name> element of the webservices.xml file for the target service.

- Set the JMS address (alternative method).

  For the SOAP over JMS provider you can instead specify the JMS address using the <soap:address> tag in the following format:

  jms:/[*queue*|*topic*]?<*property*>=<*value*>&<*property*>=<*value*>&...

  where the specification of *queue* or *topic* corresponds to the JMS address destinationStyle attribute.

Table 146. Properties that are valid for use with the <soap:address> tag. Column 1 specifies the property name, column 2 describes the property, and column 3 specifies the corresponding JMS address value.

Property name	Property description	Corresponding JMS address value
destination	The JNDI name of the destination queue or topic	jndiDestinationName
connectionFactory	The JNDI name of the connection factory.	jndiConnectionFactory
targetService	The name of the port component of the target service	targetService jms:propertyValue within jms:address
JNDI-related properties (optional):		
initialContextFactory	The name of the initial context factory.	initialContextFactory
jndiProviderURL	The JNDI provider URL	jndiProviderURL
JMS-related properties (optional):		
deliveryMode	An indication as to whether the request message should be persistent or not. The valid values are DeliveryMode.NON_PERSISTENT (default) and DeliveryMode.PERSISTENT	JMSDeliveryMode

Table 146. Properties that are valid for use with the <soap:address> tag (continued). Column 1 specifies the property name, column 2 describes the property, and column 3 specifies the corresponding JMS address value.

Property name	Property description	Corresponding JMS address value
password	The password to be used to gain access to the connection factory.	JMSPassword
priority	The JMS priority associated with the request message. Valid values are 0 to 9. The default value is 4.	JMSDeliveryMode
replyTo	The JNDI destination queue to which reply messages should be sent.	JMSReplyTo
timeToLive	The lifetime (in milliseconds) of the request message. A value of 0 indicates an infinite lifetime.	JMSTimeToLive
userid	The userid to be used to gain access to the connection factory.	JMSUserid

Here is an example of this format:

<jms:address> format:

```
<wsdl:port name="StockQuoteServicePort"

        binding="sqi:StockQuoteSoapJMSBinding">

    <jms:address destinationStyle="queue"
        jndiConnectionFactoryName="myQCF"
        jndiDestinationName="myQ"
        initialContextFactory="com.ibm.NamingFactory"
        jndiProviderURL="iiop://something:900/">

        <jms:propertyValue name="targetService"
                           type="xsd:string"
                           value="StockQuoteServicePort"/>

    </jms:address>

</wsdl:port>
```

<soap:address> format:

```
<wsdl:port name="StockQuoteServicePort"
        binding="sqi:StockQuoteSoapJMSBinding">
    <soap:address location="jms:/queue?connectionFactory=myQCF&destination
=myQ&initialContextFactory=com.ibm.NamingFactory&jndiProviderURL
=iiop://something:900/&targetService=StockQuoteServicePort" />

</wsdl:port>
```

- Set the JMS headers and properties.

  You use the <jms:property> tag to set the JMS headers and properties. This tag maps either a message part, or a literal value, into a JMS property:

  ```
  <jms:property name="Priority" {part="requestPriority" | value="fixedValue"}/>
  ```

  If the <jms:property> has a literal value, then it can also be nested within the <jms:address> tag:

  ```
  <jms:property name="Priority" value="fixedValue" />
  ```

  This form of the <jms:property> tag is also used in the native JMS binding.

Example of a WSDL that defines a SOAP over JMS binding

```
<!-- Example: SOAP over JMS Text Message -->

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        name="StockQuoteInterfaceDefinitions"
        targetNamespace="urn:StockQuoteInterface"
        xmlns:tns="urn:StockQuoteInterface"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:jms="http://schemas.xmlsoap.org/wsdl/jms/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

```
    <wsdl:message name="GetQuoteInput">
        <part name="symbol" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="GetQuoteOutput">
        <part name="value" type="xsd:float"/>
    </wsdl:message>

    <wsdl:portType name="StockQuoteInterface">
        <wsdl:operation name="GetQuote">
            <wsdl:input message="tns:GetQuoteInput"/>
            <wsdl:output message="tns:GetQuoteOutput"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="StockQuoteSoapJMSBinding" type="tns:StockQuoteInterface">
        <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/jms"/>
        <wsdl:operation name="GetQuote">
            <soap:operation soapAction="urn:StockQuoteInterface#GetQuote"/>
            <wsdl:input>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuoteServicePort"
                    binding="sqi:StockQuoteSoapJMSBinding">
            <jms:address destinationStyle="queue"
                     jndiConnectionFactoryName="myQCF"
                     jndiDestinationName="myQ"
                     initialContextFactory="com.ibm.NamingFactory"
                     jndiProviderURL="iiop://something:900/">

                     <jms:propertyValue name="targetService"
                                        type="xsd:string"
                                        value="StockQuoteServicePort"/>

            </jms:address>

        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Writing the WSDL extensions that let your WSIF service access a service at a JMS destination:

Using the native Java Message Service (JMS) provider, Web Services Invocation Framework (WSIF) clients can treat a service that is available at a JMS destination as a Web service. Use this information, and associated code fragments, to help you to write the Web Services Description Language (WSDL) extensions.

Before you begin

This topic assumes that you chose and configured a JMS provider when you installed WebSphere Application Server (either the default messaging provider, or another provider such as the WebSphere MQ messaging provider). If not, do so now as described in Choosing a messaging provider.

About this task

The WSDL extensions for JMS are identified with the namespace prefix `jms`. For example, `<jms:binding>`.

The supported operations are either one-way operations (`send` for JMS point-to-point messaging, or `publish` for JMS publish and subscribe messaging) or request-response operations (`send` and `receive` for JMS point-to-point messaging). The WSDL operations therefore specify either an input message only, or an input and an output message.

Operations that describe message interfaces with a native JMS binding do not have fault messages. No assumptions are made about the message schema or the semantics of message properties, therefore no distinction can be made between output and fault messages.

Use the following procedure, and associated code fragments, to help you to write the Web Services Description Language (WSDL) extensions that enable your WSIF service to access an underlying service at a JMS destination.

Procedure

- Set the JMS message body type.

  You use the `<jms:binding>` extension to specify the JMS message body type:

```
<wsdl:binding ... >
   <jms:binding type="messageBodyType" />
    ...
</wsdl:binding>
```

  where *messageBodyType* is either `ObjectMessage` or `TextMessage`.

- Specify the parts to use for the input and output messages.

  For JMS text messages and JMS object messages created from one or more WSDL message parts, you use the `<jms:input>` and `<jms:output>` extensions to specify the message parts to use for the JMS messages:

```
<wsdl:input ... >
   <jms:input parts="part1 part2 ..." />
</wsdl:input>

<wsdl:output ... >
   <jms:output parts="part1 part2 ..." />
</wsdl:output>
```

  In the next example, the WSDL message has just one part that contains the complete message body. This message body might result from a mapping of some other representation (see the next step “Map the data types”).

```
<wsdl:input ... >
   <jms:input parts="part1" />
</wsdl:input>
```

  If no parts are defined, then all the message parts are used.

- Map the data types.

  You use the `<format>` extensions to map data types:

```
<wsdl:binding ... >
   <jms:binding type="..." />

   <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="..." formatType="targetType"/>
   </format:typemapping>
   ...
</wsdl:binding>
```

  The value of *targetType* is dependent on the JMS message body type (see the step “Set the JMS message body type”). For JMS object messages, the target data type implements the java.io.Serializable class. For JMS text messages, the target data type is always java.lang.String.

  The `<format>` extensions are also used in other bindings that deal with Java interfaces.

- Set the JMS headers and properties.

JMS does not make assumptions about message headers. For example, if the JMS provider is MQSeries then each JMS message carries an RFH2 header. However you can access data in this message header indirectly, by getting and setting JMS message properties.

When you want your application to pass a property into the Web Services Invocation Framework (WSIF) as a part on the WSIF message, you use a <jms:property> tag. When you want to hard code a property value into the WSDL, you use a <jms:propertyValue> tag. The <jms:propertyValue> tag contains a specification of a literal value and its associated XML schema type.

You can specify <jms:property> and <jms:propertyValue> extensions within the <wsdl:input> tag in the binding operation, and also within the <jms:address> tag. For the<wsdl:output> tag in the binding operation, you can only specify the <jms:property> extension. Property values that are set in the <jms:property> tag take precedence over values set in the <<jms:propertyValue> tag, and property values that are set in the binding operation (in the <input> and <output> tags) take precedence over values set in the <jms:address> tag.

Here is an example of the <jms:property> and <jms:propertyValue> tags nested within the <input> and <output> tags:

```
<wsdl:input ... >

    <jms:property name="propertyName" part="partName" />

    <jms:propertyValue name="propertyName"
               type="xsdType" value="actualValue" />

</wsdl:input>

<wsdl:output ... >

    <jms:property name="propertyName" part="partName" />

</wsdl:output>
```

where *propertyName* identifies the JMS property that is associated with the header field, and *partName* identifies the message part that is associated with the property.

The JMS property identified by *propertyName* can be user-defined, or it can be one of the following predefined header fields:

Table 147. Predefined JMS message header fields.

Column 1 specifies the header field value, and column 2 specifies the data type for that header field value.

Value	Java type
JMSMessageId	java.lang.String
JMSTimeStamp	long
JMSCorrelationId	byte [ ] or java.lang.String
JMSReplyTo	javax.jms.Destination
JMSDestination	javax.jms.Destination
JMSDeliveryMode	int
JMSRedelivered	boolean
JMSType	java.lang.String
JMSExpiration	long
JMSTimeToLive	long

See the JMS specification for restrictions that apply when setting JMS header field values. Attempts to set restricted values are ignored.

For application-defined JMS message properties, the Java types used in the native JMS binding implementation (used for calls to the corresponding JMS methods) are derived from the XML schema type in the abstract interface (<wsdl:part> tag), and from the type mapping information in the format binding (<format:typemap> tag).

- Handle transactions.

  Independent of other JMS properties, the asynchronous processing of request-response operations has implications for callers running in a transaction scope. The send request part and the receive response part are separated into two transactions, because the send needs to be committed in order for the request message to become visible. Implementations that process WSDL for asynchronous request-response operations (such as WSIF) must therefore take the following additional actions:
  - They must ensure that the send request transaction returns a correlation ID to the user, and provides a callback with which users can pass in the response message to process the receive response transaction.
  - They might implement their own response message listener in order to recognize the arrival of response messages, and to manage the correlation to the request message.

Example 1: JMS Text Message

The JMS text message contains a java.lang.String. In this example, the WSDL message contains only one part that represents the whole message body:

```
<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest"> ... </wsdl:message>
   <wsdl:message name="JmsOperationResponse"> ... </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <jms:binding type="TextMessage" />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:String" formatType="String" />
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:input parts="requestMessageBody" />
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:output parts="responseMessageBody" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
```

```
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

As an extension to the previous JMS message example, the following example WSDL describes a request-response operation in which specific JMS property values of the request and response message are set for the request message and retrieved from the response message.

The JMS properties in the request message are set according to the values in the input message. Likewise, selected JMS properties of the response message are copied to the corresponding values of the output message. The direction of the mapping is determined by the appearance of the <jms:property> tag in the input or output section, respectively.

```
<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest">
      <wsdl:part name="myInt" type="xsd:int"/>
      ...
   </wsdl:message>

   <wsdl:message name="JmsOperationResponse">
      <wsdl:part name="myString" type="xsd:String"/>
      ...
   </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <!-- the JMS message type might be any of the above -->
      <jms:binding type="..." />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:int" formatType="int" />
         ...
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:property message="tns:JmsOperationRequest" parts="myInt" />
            <jms:propertyValue name="myLiteralString"
                           type="xsd:string" value="Hello World" />
            ...
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:property message="tns:JmsOperationResponse" parts="myString" />
            ...
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
```

```
    </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

Enabling a WSIF client to invoke a web service through JMS:

The ways in which the Web Services Invocation Framework (WSIF) interacts with the Java Message Service (JMS), and the steps to take to enable a service to be invoked through JMS by a WSIF client application.

Before you begin

This topic assumes that you chose and configured a JMS provider when you installed WebSphere Application Server (either the default messaging provider, or another provider such as the WebSphere MQ messaging provider). If not, do so now as described in Choosing a messaging provider.

About this task

Here are the ways in which WSIF interacts with JMS:
- WSIF only supports input JMS properties.
- WSIF needs two queues when invoking an operation: one for the request message and one for the reply.
- The replyTo queue is by default a temporary queue, which WSIF creates on behalf of the application. You can specify a permanent queue by setting the **JMSReplyTo** property to the JNDI name of a queue.
- WSIF uses the default values for properties set by the JMS implementation.

To enable a service to be invoked through JMS by a WSIF client application, complete the following steps:

Procedure

1. Use the administrative console to create and configure a queue connection factory and a queue destination for your chosen messaging provider.

   For more information, see Configuring resources for the default messaging provider, Configuring JMS resources for the WebSphere MQ messaging provider or Managing messaging with a third-party messaging provider.

   Note: In WebSphere MQ and some other JMS implementations, messages are persistent by default. The WSIF replyTo temporary queue is of type `temporary dynamic` by default, which means that your JMS provider cannot write a persistent response message to this queue. If you are using the WebSphere MQ messaging provider, create a temporary model queue that is of type `permanent dynamic`, then pass this model as the **tempmodel** of your queue connection factory. This ensures that persistent messages are written to a temporary replyTo queue that is of type `permanent dynamic`.
2. Use the administrative console to add the new queue destination to the list of JMS destination names for your application server. Ensure that the **Initial State** is `started`.
3. Put the JNDI names of the queue destination and queue connection factory, as well as your JNDI configuration, in the Web Services Description Language (WSDL) file.
4. Optional: If your client is running on an application server that has been migrated from WebSphere Application Server Version 5, you might get basic authentication errors and therefore have to modify your security settings. For more information see Web Services Invocation Framework troubleshooting tips.

JMS message header: The TimeToLive property reference:

The range of permitted values for the TimeToLive property of a JMS message that WSIF puts onto a queue.

The JMS message header property `JMSTimeToLive` is of type `long`. It sets the time to live of a message put onto a queue, in milliseconds. A value of `0` means live indefinitely.

The factors that determine the time to live of a JMS message are as follows:

- For a one-way (input only) operation, the default time to live is `0`, so the message remains on the queue indefinitely or until the server end-processes the message. The `JMSTimeToLive` value, if specified in the service endpoint URL or the JMS Address, is used for one-way messages. The client never waits for a response to a one-way operation and so it never times out. During a one-way operation, the client fails only if the queue itself is unavailable.
- For a two-way (request and response) operation the `JMSTimeToLive` value, if specified in the service endpoint URL or the JMS Address, is used for two-way messages. When the time to live is not specified, the default value is determined by the client response timeout setting.

Writing the WSDL extension that lets your WSIF service invoke a method on a local Java object

Using the Web Services Invocation Framework (WSIF) Java provider, WSIF can invoke Java code. This means that, in a thin-client environment such as a Java virtual machine (JVM) or Tomcat test runtime environment, you can define shortcuts to local Java programs. Use this procedure to help you to write the Web Services Description Language (WSDL) extension that links your WSIF service to a local Java application.

Before you begin

The WSIF Java provider is not intended for use in a Java Platform, Enterprise Edition (Java EE) environment. There is a difference between a client using the WSIF Java provider to invoke a Java component, and implementing a web service as a Java component on the server side.

About this task

The WSIF Java binding exploits the format binding for type mapping. Using the format binding, your WSDL can define the mapping between XML schema types and Java types.

The WSIFJava provider requires the targeted Java classes to be on the class path of the client. The Java method is invoked synchronously, in-process, in-thread, with the current thread and Object Request Broker (ORB) contexts.

The WSIF Java provider is not transactional.

The WSIF Java provider does not support the WSIF synchronous timeout. The Java provider will not time out waiting for a Java method to complete.

Use the following procedure, and associated code fragments, to help you to specify the WSDL extension that enables your WSIF service to invoke a method on a local Java object.

Procedure

- Specify the Java binding.

  To use the Java provider, you need the following binding specified in the WSDL file:

```
<!-- Java binding -->
<binding .... >
    <java:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
```

```
    <operation>*
        <java:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            methodType="instance|constructor" />
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
```

In this example:

- A question mark (?) means optional, and an asterisk (*) means 0 or more.
- The `name` attribute of the `<format:typeMap>` element is a qualified name of a simple or complex type used by one of the Java operations.
- The `formatType` attribute of the `<format:typeMap>` element is the fully qualified class name for the Java class to which the element specified by `name` maps.
- The `methodName` attribute of the `<java:operation>` element is the name of the method on the Java object that is called by the operation.
- The `parameterOrder` attribute of the `<java:operation>` element contains a white space-separated list of part names that define the order in which they are passed to the Java object method.
- The `methodType` attribute of the `<java:operation>` element must be set to either `instance` or `constructor`. The value specifies whether the method that is invoked on the object is an instance method or a constructor for the object.

- Specify the `<java:address>` element.

  The `className` attribute of the `<java:address>` element specifies the fully qualified class name of the object containing the method to invoke:

```
<service ... >
    <port>*
        <java:address
            className="nmtoken"/>
    </port>
</service>
```

Writing the WSDL extension that lets your WSIF service invoke an enterprise bean

Using the EJB provider, WSIF clients can invoke enterprise beans through Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). Use this information, and associated code fragments, to help you to write the Web Services Description Language (WSDL) extension that links your WSIF service to a service implemented as an enterprise bean.

Before you begin

Although you can use the EJB provider for EJB(IIOP)-based web service invocation, it is recommended that you instead invoke RMI-IIOP web services by using JAX-RPC.

The EJB client JAR file must be available in the client runtime environment with the current provider.

The EJB provider does not support the WSIF synchronous timeout. The EJB provider will not time out waiting for a Java method to complete.

About this task

Your WSIF client can invoke an enterprise bean by using RMI-IIOP, with the current security and transaction contexts. If the EJB provider is invoked within a transaction, the transaction is passed to the onward service and the standard EJB transaction attribute applies.

If there are multiple implementations of the service, it is up to the service providers to make sure that every implementation offers the same semantics. For example, for transactions, the bean deployer must specify TX_REQUIRES_NEW to force a new transaction.

Use the following procedure, and associated code fragments, to help you to write the Web Services Description Language (WSDL) extension that enables your WSIF service to invoke an enterprise bean.

Procedure

- Specify the EJB binding.

```
<!-- EJB binding -->
<binding .... >
    <ejb:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
    <operation>*
        <ejb:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            interface="remote|home" />
        <input name="nmtoken"? />?
        <output name="nmtoken"? />?
        <fault name="nmtoken"? />?
    </operation>
</binding>
```

  In this example:
  - A question mark (?) means optional, and an asterisk (*) means 0 or more.
  - The name attribute of the <format:typeMap> element is a qualified name of a simple or complex type used by one of the EJB operations.
  - The formatType attribute of the <format:typeMap> element is the fully qualified class name for the Java class to which the element specified by name maps.
  - The methodName attribute of the <ejb:operation> element is the name of the method on the enterprise bean that is called by the operation.
  - The parameterOrder attribute of the <ejb:operation> element contains a white space-separated list of part names that define the order in which they are passed to the EJB method.
  - The interface attribute of the <ejb:operation> element must be set to either remote or home. The value specifies the interface of the enterprise bean on which the method named by the methodName attribute is accessible.

- Specify the <ejb:address> element.

```
<service ... >
    <port>*
        <ejb:address
            className="nmtoken"
            jndiName="nmtoken"
            initialContextFactory="nmtoken" ?
            jndiProviderURL="nmtoken" ? />
    </port>
</service>
```

  In this example:
  - The className attribute of the <ejb:address> element specifies the fully qualified class name of the home interface class of the enterprise bean.
  - The jndiName attribute of the <ejb:address> element specifies the full Java Naming and Directory Interface (JNDI) name that is used to look up the enterprise bean.
  - The initialContextFactory attribute of the <ejb:address> element is optional and specifies the initial context factory class.
  - The jndiProviderURL attribute of the <ejb:address> element is optional and specifies the JNDI provider web address.

Developing a WSIF service

A Web Services Invocation Framework (WSIF) service is a web service that uses WSIF.

About this task

To develop a WSIF service, develop the web service (or use an existing web service), then develop the WSIF client based on the Web Services Description Language (WSDL) document for that Web service.

There are also two pre-built WSIF samples available for download from the WebSphere Application Server samples page of the developerWorks website:

- The Address Book sample.
- The Stock Quote sample.

For more information about using the pre-built samples, see the documentation that is included in the developerWorks download package. Note that these samples were written to work with WebSphere Application Server Version 5.

To develop a WSIF service, complete the following steps:

Procedure

1. Implement the web service.

   Use web services tools to discover, create, and publish the web service. You can develop Java bean, enterprise bean, and URL web services. You can use web service tools to create skeleton Java code and a sample application from a WSDL document. For example, an enterprise bean can be offered as a Web service, and use Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) as the access protocol. Or you can use a Java class as a web service, with native Java invocations as the access protocol.

   You can use the WebSphere Studio Application Developer to create a web service from a Java application, as described in its StockQuote service tutorial. The Java application that you use in this scenario returns the last trading price from the Internet website www.xmltoday.com, given a stock symbol. Using the web service wizard, you generate a binding WSDL document named `StockQuoteService-binding.wsdl` and a service WSDL document named `StockQuoteService-service.wsdl` from the `StockQuoteService.java` bean. You then deploy the web service to a web server, generate a client proxy to the Web service, and generate a sample application that accesses the StockQuoteService through the client proxy. You test the StockQuote web service, publish it by using the IBM UDDI Explorer, and then discover the StockQuote web service in the IBM UDDI Test Registry.

2. Develop the WSIF client.

   Use the following information to help you develop a WSIF client:

   - “Example: Using WSIF to invoke the AddressBook sample web service dynamically” on page 1386 gives example code to show how you define a web service in WSDL.
   - “Linking a WSIF service to the underlying implementation of the service” on page 1369 describes the available providers, and gives example code of how their WSDL extensions are coded.
   - “Invoking a WSDL-based web service through the WSIF API” on page 1399 defines the main interfaces that your client uses to support the invocation of Web services defined in WSDL.

   The AddressBook sample is written for synchronous interaction. If you are using a JMS provider, your WSIF client might have to act asynchronously. WSIF provides two main features that meet this requirement:

   - A correlation service that assigns identifiers to messages so that the request can match up with the (eventual) response.
   - A response handler that picks up the response from the web service at a later time.

   For more information, see “WSIFOperation - Asynchronous interactions reference” on page 1403.

Example: Using WSIF to invoke the AddressBook sample web service dynamically

This is example code for dynamic invocation of the AddressBook sample web service by using WSIF:

```
    try {
        String wsdlLocation="clients/addressbook/AddressBookSample.wsdl";

        // The starting point for any dynamic invocation using wsif is a
        // WSIFServiceFactory. Create one through the newInstance
        // method.
        WSIFServiceFactory factory = WSIFServiceFactory.newInstance();

        // Once you have a factory, you can use it to create a WSIFService object
        // corresponding to the AddressBookService service in the wsdl file.
        // Note: because you only have one service defined in the wsdl file, you
        // do not have to use the namespace and name of the service and can pass
        // null instead. This also applies to the port type, although values have
        // been used below for illustrative purposes.
        WSIFService service = factory.getService(
            wsdlLocation,    // location of the wsdl file
            null,            // service namespace
            null,            // service name
           "http://www.ibm.com/namespace/wsif/samples/ab", // port type namespace
            "AddressBookPT" // port type name
        );

        // The AddressBook.wsdl file contains the definitions for two complexType
        // elements within the schema element. Map these complexTypes
        // to Java classes. These mappings are used by the Apache SOAP provider
        service.mapType(
            new javax.xml.namespace.QName(
                "http://www.ibm.com/namespace/wsif/samples/ab/types",
                "address"),
            Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"));
        service.mapType(
            new javax.xml.namespace.QName(
                "http://www.ibm.com/namespace/wsif/samples/ab/types",
                "phone"),
            Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFPhone"));
            // You now have a WSIFService object. The next step is to create a WSIFPort
            // object for the port you want to use. The getPort(String portName) method
            // allows us to generate a WSIFPort from the port name.
        WSIFPort port = null;

        if (portName != null) {
            port = service.getPort(portName);
        }
        if (port == null) {
            // If no port name was specified, attempt to create a WSIFPort from
            // the available ports for the port type specified on the service
            port = getPortFromAvailablePortNames(service);
        }

        // Once you have a WSIFPort, you can create an operation. Execute
        // the addEntry operation and therefore attempt to create a WSIFOperation
        // corresponding to it. The addEntry operation is overloaded in the wsdl i.e.
    // there are two versions of it, each taking different parameters (parts).
    // This overloading requires that you specify the input and output message
    // names for the operation in the createOperation method so that the correct
    // operation can be resolved.
        // Because the addEntry operation has no output message, you use null for its name.
        WSIFOperation operation =
            port.createOperation("addEntry", "AddEntryWholeNameRequest", null);

        // Create messages to use in the execution of the operation. This should
        // be done by invoking the createXXXXXMessage methods on the WSIFOperation.
        WSIFMessage inputMessage = operation.createInputMessage();
```

```
WSIFMessage outputMessage = operation.createOutputMessage();
WSIFMessage faultMessage = operation.createFaultMessage();

// Create a name and address to add to the addressbook
String nameToAdd="Chris P. Bacon";
WSIFAddress addressToAdd =
    new WSIFAddress (1,
        "The Waterfront",
        "Some City",
        "NY",
        47907,
        new WSIFPhone (765, "494", "4900"));

// Add the name and address to the input message
inputMessage.setObjectPart("name", nameToAdd);
inputMessage.setObjectPart("address", addressToAdd);

// Execute the operation, obtaining a flag to indicate its success
boolean operationSucceeded =
    operation.executeRequestResponseOperation(
        inputMessage,
        outputMessage,
        faultMessage);

if (operationSucceeded) {
    System.out.println("Successfully added name and address to addressbook\n");
} else {
    System.out.println("Failed to add name and address to addressbook");
}

// Start from fresh
operation = null;
inputMessage = null;
outputMessage = null;
faultMessage = null;

// This time you will lookup an address from the addressbook.
// The getAddressFromName operation is not overloaded in the
// wsdl and therefore you can specify the operation name
// without any input or output message names.
operation = port.createOperation("getAddressFromName");

// Create the messages
inputMessage = operation.createInputMessage();
outputMessage = operation.createOutputMessage();
faultMessage = operation.createFaultMessage();

// Set the name to find in the addressbook
String nameToLookup="Chris P. Bacon";
inputMessage.setObjectPart("name", nameToLookup);

// Execute the operation
operationSucceeded =
    operation.executeRequestResponseOperation(
        inputMessage,
        outputMessage,
        faultMessage);

if (operationSucceeded) {
    System.out.println("Successful lookup of name '"+nameToLookup+"' in addressbook");

    // You can get the address that was found by querying the output message
    WSIFAddress addressFound = (WSIFAddress) outputMessage.getObjectPart("address");
    System.out.println("The address found was:");
    System.out.println(addressFound);
} else {
    System.out.println("Failed to lookup name in addressbook");
```

```
        }

    } catch (Exception e) {
        System.out.println("An exception occurred when running the sample:");
        e.printStackTrace();
    }
}
```

The preceding code refers to the following Sample method:

```
    WSIFPort getPortFromAvailablePortNames(WSIFService service)
            throws WSIFException {
        String portChosen = null;

        // Obtain a list of the available port names for the service
        Iterator it = service.getAvailablePortNames();
        {
            System.out.println("Available ports for the service are: ");
            while (it.hasNext()) {
                String nextPort = (String) it.next();
                if (portChosen == null)
                    portChosen = nextPort;
                System.out.println(" - " + nextPort);
            }
        }
        if (portChosen == null) {
            throw new WSIFException("No ports found for the service!");
        }
        System.out.println("Using port " + portChosen + "\n");

        // An alternative way of specifying the port to use on the service
        // is to use the setPreferredPort method. Once a preferred port has
        // been set on the service, a WSIFPort can be obtained through getPort
        // (no arguments). If a preferred port has not been set and more than
        // one port is available for the port type specified in the WSIFService,
        // an exception is thrown.
        service.setPreferredPort(portChosen);
        WSIFPort port = service.getPort();
        return port;
    }
```

The web service uses the following classes:

WSIFAddress:

```
public class WSIFAddress implements Serializable {

    //instance variables
    private int streetNum;
    private java.lang.String streetName;
    private java.lang.String city;
    private java.lang.String state;
    private int zip;
    private WSIFPhone phoneNumber;

    //constructors
    public WSIFAddress () { }

    public WSIFAddress (int streetNum,
                        java.lang.String streetName,
                        java.lang.String city,
                        java.lang.String state,
                        int zip,
                        WSIFPhone phoneNumber) {
          this.streetNum = streetNum;
          this.streetName = streetName;
          this.city = city;
```

```
        this.state = state;
        this.zip = zip;
        this.phoneNumber = phoneNumber;
    }

    public int getStreetNum() {
        return streetNum;
    }

    public  void setStreetNum(int streetNum) {
        this.streetNum = streetNum;
    }

    public java.lang.String getStreetName() {
        return streetName;
    }

    public  void setStreetName(java.lang.String streetName) {
        this.streetName = streetName;
    }

    public java.lang.String getCity() {
        return city;
    }

    public  void setCity(java.lang.String city) {
        this.city = city;
    }

    public java.lang.String getState() {
        return state;
    }

    public  void setState(java.lang.String state) {
        this.state = state;
    }

    public int getZip() {
        return zip;
    }

    public  void setZip(int zip) {
        this.zip = zip;
    }

    public WSIFPhone getPhoneNumber() {
        return phoneNumber;
    }

    public  void setPhoneNumber(WSIFPhone phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
}
```

WSIFPhone:

```
public class WSIFPhone implements Serializable {

    //instance variables
    private int areaCode;
    private java.lang.String exchange;
    private java.lang.String number;

    //constructors
    public WSIFPhone () { }

    public WSIFPhone (int areaCode,
```

```
                    java.lang.String exchange,
                    java.lang.String number) {
          this.areaCode = areaCode;
          this.exchange = exchange;
          this.number = number;
    }

    public int getAreaCode() {
          return areaCode;
    }

    public  void setAreaCode(int areaCode) {
          this.areaCode = areaCode;
}

    public java.lang.String getExchange() {
          return exchange;
    }

    public  void setExchange(java.lang.String exchange) {
          this.exchange = exchange;
    }

    public java.lang.String getNumber() {
          return number;
    }

    public  void setNumber(java.lang.String number) {
          this.number = number;
    }
}
```

WSIFAddressBook:

```
public class WSIFAddressBook {
    private Hashtable name2AddressTable = new Hashtable();

    public WSIFAddressBook() {
    }

    public void addEntry(String name, WSIFAddress address)
    {
        name2AddressTable.put(name, address);
    }

    public void addEntry(String firstName, String lastName, WSIFAddress address)
    {
        name2AddressTable.put(firstName+" "+lastName, address);
    }

    public WSIFAddress getAddressFromName(String name)
        throws IllegalArgumentException
    {

        if (name == null)
        {
            throw new IllegalArgumentException("The name argument must not be " +
                                               "null.");
        }
        return (WSIFAddress)name2AddressTable.get(name);
    }

}
```

The following code is the corresponding WSDL file for the web service:

```
<?xml version="1.0" ?>

<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:typens="http://www.ibm.com/namespace/wsif/samples/ab/types"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
             xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
             xmlns:ejb="http://schemas.xmlsoap.org/wsdl/ejb/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <xsd:schema
       targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab/types"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <xsd:complexType name="phone">
        <xsd:element name="areaCode" type="xsd:int"/>
        <xsd:element name="exchange" type="xsd:string"/>
        <xsd:element name="number" type="xsd:string"/>
      </xsd:complexType>

      <xsd:complexType name="address">
        <xsd:element name="streetNum" type="xsd:int"/>
        <xsd:element name="streetName" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zip" type="xsd:int"/>
        <xsd:element name="phoneNumber" type="typens:phone"/>
      </xsd:complexType>

    </xsd:schema>
  </types>

  <message name="AddEntryWholeNameRequestMessage">
    <part name="name" type="xsd:string"/>
    <part name="address" type="typens:address"/>
  </message>

  <message name="AddEntryFirstAndLastNamesRequestMessage">
    <part name="firstName" type="xsd:string"/>
    <part name="lastName" type="xsd:string"/>
    <part name="address" type="typens:address"/>
  </message>

  <message name="GetAddressFromNameRequestMessage">
    <part name="name" type="xsd:string"/>
  </message>

  <message name="GetAddressFromNameResponseMessage">
    <part name="address" type="typens:address"/>
  </message>

  <portType name="AddressBookPT">
    <operation name="addEntry">
      <input name="AddEntryWholeNameRequest"
       message="tns:AddEntryWholeNameRequestMessage"/>
    </operation>
    <operation name="addEntry">
      <input name="AddEntryFirstAndLastNamesRequest"
       message="tns:AddEntryFirstAndLastNamesRequestMessage"/>
    </operation>
    <operation name="getAddressFromName">
      <input name="GetAddressFromNameRequest"
       message="tns:GetAddressFromNameRequestMessage"/>
      <output name="GetAddressFromNameResponse"
```

```
      message="tns:GetAddressFromNameResponseMessage"/>
  </operation>
</portType>

<binding name="SOAPHttpBinding" type="tns:AddressBookPT">
  <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryWholeNameRequest">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryFirstAndLastNamesRequest">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="getAddressFromName">
    <soap:operation soapAction=""/>
    <input name="GetAddressFromNameRequest">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output name="GetAddressFromNameResponse">
      <soap:body use="encoded"
                namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>

<binding name="JavaBinding" type="tns:AddressBookPT">
  <java:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
           formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="name address"
       methodType="instance"/>
    <input name="AddEntryWholeNameRequest"/>
  </operation>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="firstName lastName address"
       methodType="instance"/>
    <input name="AddEntryFirstAndLastNamesRequest"/>
  </operation>
  <operation name="getAddressFromName">
    <java:operation
       methodName="getAddressFromName"
       parameterOrder="name"
       methodType="instance"
       returnPart="address"/>
    <input name="GetAddressFromNameRequest"/>
    <output name="GetAddressFromNameResponse"/>
```

```
    </operation>
  </binding>

  <binding name="EJBBinding" type="tns:AddressBookPT">
    <ejb:binding/>
    <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="typens:address"
             formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
      <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
    </format:typeMapping>
    <operation name="addEntry">
      <ejb:operation
         methodName="addEntry"
         parameterOrder="name address"
         interface="remote"/>
      <input name="AddEntryWholeNameRequest"/>
    </operation>
    <operation name="addEntry">
      <ejb:operation
         methodName="addEntry"
         parameterOrder="firstName lastName address"
         interface="remote"/>
      <input name="AddEntryFirstAndLastNamesRequest"/>
    </operation>
    <operation name="getAddressFromName">
      <ejb:operation
         methodName="getAddressFromName"
         parameterOrder="name"
         interface="remote"
         returnPart="address"/>
      <input name="GetAddressFromNameRequest"/>
      <output name="GetAddressFromNameResponse"/>
    </operation>
  </binding>
  <service name="AddressBookService">
    <port name="SOAPPort" binding="tns:SOAPHttpBinding">
      <soap:address
       location="http://myServer/wsif/samples/addressbook/soap/servlet/rpcrouter"/>
    </port>
    <port name="JavaPort" binding="tns:JavaBinding">
      <java:address className="services.addressbook.WSIFAddressBook"/>
    </port>
    <port name="EJBPort" binding="tns:EJBBinding">
      <ejb:address className="services.addressbook.ejb.AddressBookHome"
        jndiName="ejb/samples/wsif/AddressBook"
  classLoader="services.addressbook.ejb.AddressBook.ClassLoader"/>
    </port>
  </service>

</definitions>
```

Using complex types

WSIF supports user-defined complex types through the mapping of complex types to Java classes. You can specify this mapping manually or automatically.

About this task

Any calls to the WSIFService mapType and mapPackage methods used for manual mapping override any equivalent mapping information that is produced automatically. This override helps to maintain backwards compatibility, and also accommodates less standard mappings.

To map your user-defined complex types to Java classes, complete either of the following steps:

- Manually map complex types.
- Automatically map complex types.

Procedure

- Manually map complex types.

  The method to use when you create these mappings manually depends on the provider. For the Java and EJB providers, the mappings are specified in the Web Services Description Language (WSDL) file in the binding element. The following example provides the syntax for specifying the mapping:

```
<binding .... >
    <ejb:binding|java:binding/>
      <format:typeMapping style="Java" encoding="Java"/>?
          <format:typeMap typeName="qname" formatType="nmtoken"/>*
      </format:typeMapping>
 ...
</binding>
```

  In this example:

  – A question mark (“?”) means “optional” and an asterisk (“*”) means “0 or more”.

  – The format:typeMap **typeName** attribute is a qualified name of a complex type or simple type used by one of the operations.

  – The format:typeMap **formatType** attribute is the fully qualified class name for the Java class to which the element specified by **typeName** maps.

  If you use the Apache SOAP provider then you specify the mapping of a complex type to a Java class in the client code through two methods on the org.apache.wsif.WSIFService interface:

```
public void mapType(QName elementType, Class javaType)
```

  and

```
public void mapPackage(String namespaceURI, String packageName)
```

  Use the **mapType** method to specify a mapping between an XML schema element and a Java class. The method takes a QName representing the complex type or simple type, and the corresponding Java class to which it maps.

  Use the **mapPackage** method to specify a more general mapping between a namespace and a Java package. Any custom, complex or simple type whose namespace matches that of the mapping is mapped to a Java class in the corresponding package. The name of the class is derived from the name of the complex type using standard XML to Java naming conventions.

- Automatically map complex types.

  For complex types defined in the WSDL, where a generated bean is used to represent this type in Java, the Web Services Invocation Framework (WSIF) programming model requires that a call is made to the WSIFService.mapType() method. This call tells WSIF the package and class name of the bean representing the XML schema type that is identified with a QName. To make things easier, the WSIFService.mapPackage() method provides a mechanism to specify a wildcard version of this, where any class within a specified package is mapped to the namespace of a QName. This is a mechanism for manually mapping an XML schema type to a Java class and back again (one mapping entry provides a bidirectional mapping).

  There are many ways to convert a QName representing an XML schema type name to a Java package name and class. To enable automatic type mapping, set the WSIF_FEATURE_AUTO_MAP_TYPES feature on the WSIFServiceFactory instance:

```
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
factory.setFeature(WSIFConstants.WSIF_FEATURE_AUTO_MAP_TYPES, new Boolean(true));
```

  WSIF maps types by converting the URI part of the XML schema type QName to a package name, and converting the local part to a class name. WSIF does this mapping by using the WSIFUtils methods getPackageNameFromNamespaceURI and getJavaClassNameFromXMLName.

Using WSIF to bind a JNDI reference to a web service

You can use the Web Services Invocation Framework (WSIF) to bind a reference to a web service, then look up the reference by using JNDI.

About this task

You access a web service through information provided in the Web Services Description Language (WSDL) document for the service. If you do not know where to find the WSDL document for the service, but you know that it has been registered in a UDDI registry, then you look it up in the registry. Java programs access Java objects and resources in a similar manner, but using a JNDI interface.

The code fragments in the following steps show how, by using WSIF, you can bind a reference to a web service then look up the reference by using JNDI.

Procedure

- Specify the argument values for the web service.

  The web service is represented in WSIF by an instance of the org.apache.wsif.naming.WSIFServiceRef class. This simple Referenceable object has the following constructor:

```
public WSIFServiceRef(
        String WSDL,
        String sNS,
        String sName,
        String ptNS,
        String ptName)
{
    [...]
}
```

  In this example
  - *WSDL* is the location of the WSDL file containing the definition of the service.
  - *sNS* is the full namespace for the service definition (you can specify `null` if only one service is defined in the WSDL file).
  - *sName* is the local name for the service definition (you can specify `null` if only one service is defined in the WSDL file).
  - *ptNS* is the full namespace for the port type within the service that you want to use (you can specify `null` if only one port type is available for the service).
  - *ptName* is the local name for the port type (you can specify `null` if only one port type is available for the service).

  For example, if the WSDL file for the web service is available from the web address `http://myServer/WSDL/Example.WSDL` and contains the following service and port type definitions:

```
  <definitions targetNamespace="http://hostname/namespace/example"
               xmlns:abc="http://hostname/namespace/abc"
[...]
    <portType name="ExamplePT">
      <operation name="exampleOp">
        <input name="exampleInput" message="tns:ExampleInputMsg"/>
      </operation>
    </portType>
[...]
    <service name="abc:ExampleService">
[...]
    </service>
[...]
  </definitions>
```

  You can specify the following argument values for the WSIFServiceRef class:
  - *WSDL* is `http://myServer/WSDL/Example.WSDL`
  - *sNS* is `http://hostname/namespace/abc`
  - *sName* is `ExampleService`
  - *ptNS* is `http://hostname/namespace/example`
  - *ptName* is `ExamplePT`
- Bind the service by using JNDI.

To bind the service reference in the naming directory by using JNDI, you can use the com.ibm.websphere.naming.JndiHelper class in WebSphere Application Server:

```
[...]
    import com.ibm.websphere.naming.JndiHelper;
    import org.apache.wsif.naming.*;
[...]
    try {
       Context startingContext = new InitialContext();
       WSIFServiceRef ref = new WSIFServiceRef("http://myServer/WSDL/Example.WSDL",
                                               "http://hostname/namespace/abc"
                                               "ExampleService",
                                               "http://hostname/namespace/example",
                                               "ExamplePT");
       JndiHelper.recursiveRebind(startingContext,
            "myContext/mySubContext/myServiceRef", ref);

    }
    catch (NamingException e) {
       // Handle  error.
    }
[...]
```

- Look up the service by using JNDI.

  The following code fragment shows the lookup of a service by using JNDI:

```
[...]
    try {
[...]
       InitialContext ic = new InitialContext();
       WSIFService myService =
        (WSIFService) ic.lookup("myContext/mySubContext/myServiceRef");
[...]
    }
    catch (NamingException e) {
       // Handle error.
    }
[...]
```

Example: Passing SOAP messages with attachments by using WSIF

Information and example code for using the Web Services Invocation Framework (WSIF) SOAP provider to pass attachments within a MIME multipart/related message in such a way that the SOAP processing rules for a standard SOAP message are not changed. This includes how to write the Web Services Description Language (WSDL) extensions for SOAP attachments, and how to work with types and type mappings.

The W3C SOAP Messages with Attachments document describes a standard way to associate a SOAP message with one or more attachments in their native format (for example GIF or JPEG) by using a multipart MIME structure for transport. It defines specific use of the “Multipart/Related” MIME media type, and rules for the use of URI references to entities bundled within the MIME package. It thereby outlines a technique for carrying a SOAP 1.1 message within a MIME multipart/related message in such a way that the SOAP processing rules for a standard SOAP message are not changed.

WSIF supports passing attachments in a MIME message using the SOAP provider to link a WSIF service to a SOAP over HTTP service. The attachment is a javax.activation.DataHandler object. The mime:multipartRelated, mime:part and mime:content tags are used to describe the attachment in the WSDL.

- “Example: Writing the WSDL extensions for SOAP attachments” on page 1397
- “Example: Using WSIF to pass SOAP attachments” on page 1397
- “SOAP attachments - Working with types and type mappings” on page 1398
- “SOAP attachments - scenarios that are not supported” on page 1398

Example: Writing the WSDL extensions for SOAP attachments

The following example WSDL illustrates a simple operation that has one attachment called `attch`:

```
<binding name="MyBinding" type="tns:abc" >
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="MyOperation">
    <soap:operation soapAction=""/>
    <input>
      <mime:multipartRelated>
        <mime:part>
          <soap:body use="encoded" namespace="http://mynamespace"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </mime:part>
        <mime:part>
          <mime:content part="attch" type="text/html"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
  </operation>
</binding>
```

In this type of WSDL extension:

- There must be a `part` attribute (in this example `attch`) on the input message for the operation (in this example `MyOperation`). There can be other input parts to `MyOperation` that are not attachments.
- In the binding input there must either be a `<soap:body>` tag or a `<mime:multipartRelated>` tag, but not both.
- For MIME messages, the `<soap:body>` tag is inside a `<mime:part>` tag. There must only be one `<mime:part>` tag that contains a `<soap:body>` tag in the binding input and that must not contain a `<mime:content>` tag as well, because a content type of `text/xml` is assumed for the `<soap:body>` tag.
- There can be multiple attachments in a MIME message, each described by a `<mime:part>` tag.
- Each `<mime:part>` tag that does not contain a `<soap:body>` tag contains a `<mime:content>` tag that describes the attachment itself. The `type` attribute inside the `<mime:content>` tag is not checked or used by the Web Services Invocation Framework (WSIF). It is there to suggest to the application that uses WSIF what the attachment contains. Multiple `<mime:content>` tags inside a single `<mime:part>` tag means that the backend service expects a single attachment with a type specified by one of the `<mime:content>` tags inside that `<mime:part>` tag.
- The `parts="..."` attribute (optional) inside the `<soap:body>` tag is assumed to contain the names of all the MIME parts as well as the names of all the SOAP parts in the message.

Example: Using WSIF to pass SOAP attachments

The following code fragment can invoke the service described by the example WSDL in “Example: Writing the WSDL extensions for SOAP attachments”:

```
import javax.activation.DataHandler;
. . .
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
WSIFOperation op = service.getPort().createOperation("MyOperation");
WSIFMessage in = op.createInputMessage();
in.setObjectPart("attch",dh);
op.executeInputOnlyOperation(in);
```

The associated type mapping in the `DeploymentDescriptor.xml` file depends upon your SOAP server. For example if you use Tomcat with SOAP 2.3, then the `DeploymentDescriptor.xml` file contains the following type mapping:

```
<isd:mappings>
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:x="http://mynamespace"
 qname="x:datahandler"
```

```
 javaType="javax.activation.DataHandler"
 java2XMLClassName="org.apache.soap.encoding.soapenc.MimePartSerializer"
 xml2JavaClassName="org.apache.soap.encoding.soapenc.MimePartSerializer" />
</isd:mappings>
```

In this case, the backend service is invoked with the following signature:

```
public void MyOperation(DataHandler dh);
```

You can also use stubs to pass attachments into the Web Services Invocation Framework (WSIF):

```
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
MyInterface stub = (MyInterface)service.getStub(MyInterface.class);
stub.MyOperation(dh);
```

Attachments can also be returned from an operation, but only one attachment can be returned as the return parameter.

SOAP attachments - Working with types and type mappings

By default, attachments are passed into WSIF as DataHandler objects. If the part on the message that is the DataHandler object maps to a `<mime:part>` tag in the WSDL, then WSIF automatically maps the fully qualified name of the WSDL type to the DataHandler class and sets up that type mapping with the SOAP provider.

In your WSDL, you might have defined a schema for the attachment (for instance as a `binary[]` type). WSIF silently ignores this mapping and treats the attachment as a DataHandler object, unless you explicitly issue a `mapType()` method. WSIF lets the SOAP provider set the MIME content type based on the type of the DataHandler object, instead of the `type` attribute specified for the `<mime:content>` tag in the WSDL.

SOAP attachments - scenarios that are not supported

The following scenarios are not supported:
- Using DIME.
- Passing in `javax.xml.transform.Source` and `javax.mail.internet.MimeMultipart`.
- Using the mime:mimeXml WSDL tag.
- Nesting a mime:multipartRelated tag inside a mime:part tag.
- Using types that extend `DataHandler`, `Image`, and so on.
- Using types that contain `DataHandler`, `Image`, and soon.
- Using Arrays or Vectors of `DataHandlers`, `Images`, and so on.
- Using multiple in/out or output attachments.

The MIME headers from the incoming message are not preserved for referenced attachments. The outgoing message contains new MIME headers for Content-Type, Content-Id and Content-Transfer-Encoding that are created by WSIF.

Interacting with the Java EE container in WebSphere Application Server

How, and to what extent, WSIF interacts with the Java EE container that is provided in WebSphere Application Server.

About this task

You can interact with a container in any of the following ways:

Procedure

- Use the application server administrative console to define web services to WebSphere Application Server. This task is described in “Using WSIF to bind a JNDI reference to a web service” on page 1394. As part of the definition of a service, the administrator might define a “preferred port”.
- Use the Web Services Invocation Framework (WSIF) to make log and trace calls to the JRAS services in WebSphere Application Server, as described in Tracing and logging WSIF.
- Use WSIF providers to access Java Platform, Enterprise Edition (Java EE) services. For example, use the EJB provider to access the Java Naming and Directory Interface (JNDI) and make calls to remote enterprise beans.
- Use WSIF to wrap the use of container services so that, when WSIF is run in an unmanaged (thin) environment, the operation can succeed.

Invoking a WSDL-based web service through the WSIF API

The Web Services Invocation Framework (WSIF) provides a Java API for invoking web services, independent of the format of the service, or the transport protocol through which it is invoked.

Before you begin

WSIF includes an EJB provider for EJB invocation that uses the Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP). However, for EJB(IIOP)-based web service invocation, invoke RMI-IIOP Web services using JAX-RPC instead.

Ensure that your application uses only one thread to call WSIF.

About this task

The WSIF API supports the invocation of Web Services Description Language (WSDL)-defined web services. WSIF is intended for use in both WSIF clients and web service intermediaries.

The WSIF API is driven by the abstract service description in WSDL; it is completely independent of the binding used. This independence makes the API more natural to work with because it uses WSDL terms to refer to message parts, operations, and other items.

The WSIF API was designed for the WSDL usage model:

1. Select a port that supports the port type that you need.
2. Invoke the operation by providing the necessary abstract input message consisting of the required parts, without worrying about how the message is mapped to a specific binding protocol.

Other web service APIs, for example SOAP APIs, are not designed on WSDL, but for a specific binding protocol with its associated syntax; for example, target URIs and encoding styles.

The main WSIF API interfaces are described in the following procedure. For additional technical details of the WSIF API, see the generated API documentation that is supplied with WSIF (see the Apache WSIF website).

Procedure

- Create a message for sending to a port through the WSIFMessage interface.

  In WSDL, a message describes the abstract type of the input or output to an operation. The corresponding WSIF class is WSIFMessage, which represents in memory the input or output of an operation. The WSIFMessage interface separates the representation of the data from the abstract type defined by WSDL.

  A WSIFMessage class is a container for a set of named parts. WSIFMessage classes can be sent between Java Virtual Machines (JVMs).

1. Choose whether to represent the WSIF message at run time as a Java class or as XML.

   There are two natural ways to represent a WSDL message in a runtime environment:
   – The generated Java class, based on a WSDL to Java mapping such as that provided by a Java API for XML-based remote procedure call (JAX-RPC).
   – The XML representation of the data, for example using SOAP Encoding.

   Each option offers benefits in different scenarios. The Java class is the natural approach when WSIF is used in a standard Java client. However, in other scenarios where WSIF is used in an intermediary, it might be more efficient to keep a WSDL message in the SOAP encoded format. The style used to define messages must be consistent within the message, so all the parts in one message must be consistent. A string - `getRepresentationStyle()` - always returns `null`. This indicates that parts on this WSIFMessage class are represented as Java objects.

2. Get and set the parts of the WSIF message.

   You add parts to a WSIFMessage class with the setObjectPart or set*Type*Part methods. Each part is named. Part names within a message are unique. If you set a part more than once, the last setting is the one that is used.

   You retrieve parts by name from a WSIFMessage class with the getObjectPart or get*Type*Part methods. If the named part does not exist, the method returns a WSIFException error.

   You can use Iterators to retrieve parts from the message through the getParts() and getPartNames() methods.

   The order in which you set the parts is not important, but the message implementation might be more efficient if the parts are set in the parameter order specified by WSDL.

   WSIFMessage classes are cloneable and serializable. If the parts set are not cloneable, the implementation can try to clone them using serialization. If the parts are not serializable either, then a CloneNotSupportedException exception is thrown if cloning is attempted.

3. Set the WSIF message name.

   In addition to the containing parts, a WSIFMessage class also has a message name. This is required for operation overloading, which is supported by WSDL and WSIF.

For more information about the WSIFMessage interface (/wsi/org/apache/wsif/WSIFMessage.html) see the generated API documentation that is supplied with WSIF.

- Find a port factory or service through the WSIFService interface and the WSIFServiceFactory class.

  The WSIFService interface is a port factory that models and supports the WSDL approach in which a service is available on one or more ports. The factory hides the implementation of the port. WSIF supports dynamic ports that are based on a particular protocol and transport and configured using the WSDL at run time. For example, the dynamic SOAP port can invoke any SOAP service based on the WSDL description of that service, so you can hide and modify the set of available ports at run time.

  To find a service from a WSDL document at a web address, or from a code-generated code base, use the WSIFServiceFactory class.

- Invoke an operation through the WSIFPort interface and the WSIFOperation interface.

  A WSIFPort interface handles the details of invoking an operation. The port provides access to the implementation of the service.

  A WSDL document can provide many different WSDL bindings, and these bindings can drive multiple ports. The client can choose a port, the service stub can choose a port, or WSIF can choose a default port.

  The port offers an interface to retrieve an Operation object. A WSIFOperation interface offers the ability to run the given operation.

  If the port is serialized and deserialized at a later time, WSIF ensures that the client provides the correct information to the server to identify the instance. If the server instance is no longer available, the server must decide whether to create a fault or provide a new instance. That behavior can depend on the type of service. For example, for an enterprise bean, the WSIFPort interface stores the EJB Home and uses

it to select the bean before each invocation. The client is responsible for serializing or maintaining the port instance if it requires instance support. The client must create a new operation and messages for each invocation.

WSIFService interface

The WSIFService interface can generate an instance of the WSIFOperation interface to use for a particular invocation of a service operation.

The Web Services Invocation Framework (WSIF) service stores a list of providers that can each generate a WSIF operation for a particular Web Services Description Language (WSDL) binding. This service looks up providers by the provider type. For example, the service knows about one provider that handles SOAP ports, and other providers that handle Java ports that you define. In a managed environment, the container can configure the WSIFService interface.

For more information about the WSIFService interface, see the generated API documentation that is supplied with WSIF (see the Apache WSIF website).

A WSIFService implementation can choose a preferred port based on a number of criteria. The WSIFService implementation can set the preferred port, or it can be set by calling the setPreferredPort method.

The getPort method returns an instance of the WSIFPort class that is used to invoke a service on the port. Variants of the getPort method are used to define the characteristics of the port to be created:

- the getPort method with no arguments returns the preferred port.
- the getPort method with a string argument returns the port named by the string containing the WSDL identifier for the selected port.

The return value is `null` if the port name is not valid.

If a port is chosen (either by the WSIFService implementation, or by the setPreferredPort method), then the WSIFService implementation validates that the relevant provider exists and is configured. If the provider fails this validation check, the WSIFService interface chooses any other port for which a provider is defined. For example, if the preferred port is SOAP over JMS but the JMS libraries are not available, then WSIF chooses another port. If no preferred port is set, or the preferred port is not available, the WSIF implementation chooses the first available port listed in the WSDL.

The getAvailablePortNames() method returns, as an iteration of strings, the list of WSDL port names filtered by the set of available providers.

The getDefinition() method returns the WSDL definition for the service. If the WSDL definition is not available, this method returns `null`.

WSIFServiceFactory class

To find a service from a Web Services Description Language (WSDL) document at a web address, or from a code-generated code base, you can use the WSIFServiceFactory class.

Note: When you create a WSIFService interface from a WSIFServiceFactory class, you can specify a ClassLoader object to use in locating the WSDL file. You must specify this object when the WSDL file is in a JAR file. In such a case, specify the location of the WSDL file relative to the root of the JAR file, and use forward slashes (/) with the preceding slash removed.

For example:

```
com/myCompany/wsdl/MyWSDLFile.wsdl
```

rather than

```
/com/myCompany/wsdl/MyWSDLFile.wsdl
```

For more information about the WSIFServiceFactory class, see the generated API documentation that is supplied with WSIF (see the Apache WSIF website).

The WSIFServiceFactory class returns `null` if no service is found with that identifier.

WSIFPort interface

The port implements a factory method for the WSIFOperation interface.

For detailed information about the WSIFPort interface, see the generated API documentation that is supplied with WSIF (see the Apache WSIF website).

The createOperation(String) method returns a new instance of a WSIFOperation object. If the operationName value is not valid, or the operation is overloaded, the method throws an exception.

The createOperation(String, String, String) method supports overloaded Web Services Description Language (WSDL) operations. You can overload based on the input parameters, but not on the output parameters.

The client must call the close method when a port is no longer in use. In many cases, where the transport is sessionless, such as HTTP, this has no effect. However, if the port is using a session-based protocol such as MQSeries, Java Message Service (JMS), or External Call Interface (ECI), this supports the port in caching an open connection to the server and then closing it as required. Responsibly-written applications will call the close method if appropriate.

WSIFOperation interface

You use the WSIFOperation interface to invoke a service, based on a particular binding.

The WSIFOperation interface is the runtime representation of an operation. This interface provides methods to create input, output, and fault messages, and to invoke the operation.

For more information about the WSIFOperation interface, see the generated API documentation that is supplied with WSIF (see the Apache WSIF website).

createInputMessage, createOutputMessage and createFaultMessage
: These are factory methods to create the messages required by the invocation methods. All invocation methods require an input message.

executeRequestResponseOperation
: This method invokes “In Out” operations.

executeInputOnlyOperation
: This method invokes “In only” operations.

executeRequestResponseOperation
: If this method is used for invocation, an output and a fault message are instantiated and passed on the call to the method. If the method returns `true`, the output message contains the response message. If the message returns `false`, a fault occurred and is returned in the fault message.

executeRequestResponseAsync
: This method allows “In Out” operations to be invoked with the reply handled by using an alternative thread. Use of this method is discussed further in “WSIFOperation - Asynchronous interactions reference” on page 1403.

setContext and getContext
: Use of these methods is discussed in “WSIFOperation - Context” on page 1403.

All of the **execute*Nnnn*** methods fail with an exception if there is an error in processing the request in the WSIF provider.

Setting the timeouts for synchronous and asynchronous operations is discussed in “WSIFOperation - Synchronous and asynchronous timeouts reference” on page 1404.

WSIFOperation - Context:

Although Web Services Description Language (WSDL) does not define context, a number of uses of the Web Services Invocation Framework (WSIF) require the ability to pass context to the port that is invoking the service.

For example, a SOAP over HTTP port might require an HTTP user name and password. This information is specific to the invocation, but is not a parameter of the service. In general, context is defined as a set of name-value pairs. However, because web services tend to define the types of data by using XML schema types, WSIF represents the name-value pairs of the context by using the same representation that WSIFMessage classes use; that is, a set of named parts, each of which equates to an instance of an XML schema type.

You use the WSIFOperation interface setContext and getContext methods to pass context information to the binding. The port implementation can use this context, for example to update a SOAP header. There is no definition of how a port can use the context.

The parameter of the setContext and getContext methods is a WSIFMessage interface, and this interface has named parts defining the context information. The WSIFConstants class defines constants for the part names that can be set in a context WSIFMessage interface.

The following code fragment shows how to set the user name and password for HTTP basic authentication:

```
// set a basic authentication header
   WSIFMessage headers = new WSIFDefaultMessage();
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_USER, "user name" );
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_PSWD, "password" );
   operation.setContext( headers );
```

The WSIFOperation interface ignores context parts that it does not support. For example, the previous code is ignored by the WSIF Java provider.

The WSIFConstants class includes the following constants that can be used for context part names:
- CONTEXT_HTTP_USER
- CONTEXT_HTTP_PSWD
- CONTEXT_SOAP_HEADERS

The HTTP header values are expected to be of type String, and the SOAP header value is expected to be of type java.util.List, which should contain entries of type org.w3c.dom.Element.

WSIFOperation - Asynchronous interactions reference:

The Web Services Invocation Framework (WSIF) supports asynchronous operation. In this mode of operation, the client puts the request message as part of one transaction, and carries on with the thread of execution. The response message is then handled by a different thread, with a separate transaction.

Asynchronous operation is supported by the WSIF providers for SOAP over JMS and native JMS.

The WSIFPort class uses the supportsAsync method to test whether asynchronous operation is supported.

An asynchronous operation is initiated with the WSIFOperation interface executeRequestResponseAsync method. This method lets a Remote Procedure Call (RPC) method be invoked asynchronously. The method returns before the operation is completed, and the thread of execution continues.

The response to the asynchronous request is processed by the WSIFOperation interface fireAsyncResponse or processAsyncResponse methods.

To initiate the request, there are two forms of the executeRequestResponseAsync method:

- `public WSIFCorrelationId executeRequestResponseAsync (WSIFMessage input, WSIFResponseHandler handler)`
- `public WSIFCorrelationId executeRequestResponseAsync (WSIFMessage input)`

executeRequestResponseAsync(WSIFMessage input, WSIFResponseHandler handler)

This method takes an input message and a WSIFResponseHandler handler. The handler is invoked on another thread when the operation completes. When using this method, the client listener calls the fireAsyncResponse method, which then calls the WSIFResponseHandler interface executeAsyncResponse method.

For more information about the WSIFResponseHandler interface, see the generated API documentation that is supplied with WSIF (see the Apache WSIF website).

executeRequestResponseAsync(WSIFMessage input)

This method only takes an input message, and does not use a response handler. The client listener processes the response by calling the WSIFOperation interface processAsyncResponse method. This process updates the WSIFMessage output and fault messages with the result of the request.

WSIF supports correlation between the asynchronous request and response. When the request is sent, the WSIFOperation object is serialized into the WSIFCorrelationService object. The executeRequestResponseAsync methods return a WSIFCorrelationId object that identifies the serialized WSIFOperation object. The client listener can use this to match a response to a particular request.

The correlation service is located with the WSIFCorrelationServiceLocator class getCorrelationService() method in the org.apache.wsif.utils package.

In a managed container a default correlation service is defined in the default Java Naming and Directory Interface (JNDI) namespace by using the name: java:comp/wsif/WSIFCorrelationService. If this correlation service is not available, WSIF uses the WSIFDefaultCorrelationService.

For more information about the WSIFCorrelationService interface, see the generated API documentation that is supplied with WSIF.

This is the correlator ID:

```
public interface WSIFCorrelator extends Serializable {
   public String getCorrelationId();
}
```

The client must implement its own response message listener or message data base so that it can recognize the arrival of response messages. This client implementation manages the correlation of the response message to the request and call of one of the asynchronous response processing methods. As an example of the requirement for a client listener, the following code fragment shows what can be in the onMessage method of a Java Message Service (JMS) listener:

```
public void onMessage(Message msg) {
     WSIFCorrelationService cs = WSIFCorrelationServiceLocator.getCorrelationService();
           WSIFCorrelationId cid = new JmsCorrelationId( msg.getJMSCorrelationID() );
           WSIFOperation op = cs.get( cid );
           op.fireAsyncResponse( msg );
}
```

WSIFOperation - Synchronous and asynchronous timeouts reference:

When you use the Web Services Invocation Framework (WSIF) with the Java Message Service (JMS) you can set timeouts for synchronous and asynchronous operations.

Default values for these timeouts are defined in the wsif.properties file:

```
# maximum number of milliseconds to wait for a response to a synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined or invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

If you use these default values, a synchronous request (such as a WSIFOperation interface executeRequestResponseOperation method call) times out after ten seconds, and an asynchronous request (such as a WSIFOperation interface executeRequestResponseAsync method call) times out after sixty seconds.

Note:

> The code that processes both of these timeout values uses milliseconds as its unit of time. The WSIFProperties class getAsyncTimeout method multiplies the wsif.asyncrequest.timeout value by 1000, to convert the value from seconds to milliseconds.

You can override these default values for a given request by writing a WSDL extension that sets a JMS property on the operation request with the `<jms:property>` and `<jms:propertyValue>` WSDL elements. Set the name of the property to be the name of the timeout from the WSIF properties file.

The following example sets synchronous requests to time out after two minutes (120 seconds):

```
<jms:propertyValue name="wsif.syncrequest.timeout" type="xsd:string" value="120000"/>
```

and the following example disables asynchronous timeouts (a value of zero means wait forever):

```
<jms:propertyValue name="wsif.asyncrequest.timeout" type="xsd:string" value="0"/>
```

When an asynchronous timeout expires, no listener or message data base waiting for the response is notified. The asynchronous timeout is only used to tell the correlation service that the stored WSIFOperation can be deleted. For more information about the correlation service, see “WSIFOperation - Asynchronous interactions reference” on page 1403.

Running WSIF as a client

You can run the Web Services Invocation Framework (WSIF) in the Application Client for WebSphere Application Server, and in similar clients from other suppliers.

Procedure

To simplify the process of launching client applications in the Application Client for WebSphere Application Server, use the `launchClient` tool as described in “Running a Java EE client application with launchClient” on page 2015.

Chapter 31. Developing web services - Notification (WS-Notification)

WS-Notification enables web services to use the publish and subscribe messaging pattern.You use publish and subscribe messaging to publish one message to many subscribers. In this pattern a producing application inserts (publishes) a message (event notification) into the messaging system having marked it with a topic that indicates the subject area of the message. Consuming applications that have subscribed to the topic in question, and have appropriate authority, all receive an independent copy of the message that was published by the producing application.

Developing applications that use WS-Notification

You can code a single application to undertake several WS-Notification tasks. These topics provide sample code for common tasks that your WS-Notification applications can perform.

Before you begin

Most of these examples use the Java API for XML-based remote procedure call (JAX-RPC) APIs and WebSphere Application Server APIs and SPIs. These JAX-RPC examples can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

If you have an existing Version 6.1 WS-Notification configuration, and you want to use WS-Notification with policy sets, work through the following tasks:

1. Migrating a Version 6.1 WS-Notification configuration from WebSphere Application Server Version 6.1 to Version 7.0 or later.
2. Preparing a migrated Version 6.1 WS-Notification configuration for reliable notification.
3. Configuring WS-Notification for reliable notification.

Your applications can also use WS-Notification to receive event notifications generated by other clients of the service integration bus such as JMS clients. This is described in Topology for WS-Notification as an entry or exit point to the service integration bus and Providing access for WS-Notification applications to an existing bus topic space. For information about developing applications for a mixed clients solution, including cross-streaming from a JMS client, see Interacting with JMS message types.

About this task

A single application can be coded to undertake several WS-Notification tasks. Use the examples to help you code these tasks into your WS-Notification applications.

For an overview of how applications can use the notification broker, see WS-Notification: How client applications interact at runtime.

WS-Notification applications divide into two broad types: those that expose a Web service endpoint (for example a WS-Notification consumer application that receives notifications of stock valuation changes),

and those that do not expose a web service endpoint (for example applications that generate notifications of stock valuation changes). For broad guidance on the steps you take to develop each of these application types, see the following topics:

- "Writing a WS-Notification application that exposes a web service endpoint."
- "Writing a WS-Notification application that does not expose a web service endpoint" on page 1409.

Rather than receiving all messages on a topic to which you have subscribed, your consuming application can use XML Path (XPath) selectors to filter the messages based upon the contents of each message as described in "Filtering the message content of publications" on page 1410.

The code examples listed in this topic use the following WebSphere Application Server APIs and SPIs:

```
com.ibm.websphere.sib.wsn.AbsoluteOrRelativeTime;
com.ibm.websphere.sib.wsn.CreatePullPoint;
com.ibm.websphere.sib.wsn.CreatePullPointResponse;
com.ibm.websphere.sib.wsn.Filter;
com.ibm.websphere.sib.wsn.GetMessages;
com.ibm.websphere.sib.wsn.GetMessagesResponse;
com.ibm.websphere.sib.wsn.NotificationMessage;
com.ibm.websphere.sib.wsn.TopicExpression;
com.ibm.websphere.webservices.soap.IBMSOAPFactory;
com.ibm.websphere.wsaddressing.EndpointReference;
com.ibm.websphere.wsaddressing.WSAConstants;
com.ibm.wsspi.wsaddressing.EndpointReferenceManager;
```

Procedure

- For examples of client applications, see the following topics:
  1. "Example: Subscribing a WS-Notification consumer" on page 1411.
  2. "Example: Pausing a WS-Notification subscription" on page 1414.
  3. "Example: Publishing a WS-Notification message" on page 1415.
  4. "Example: Creating a WS-Notification pull point" on page 1417.
  5. "Example: Getting messages from a WS-Notification pull point" on page 1418.
  6. "Example: Registering a WS-Notification publisher" on page 1419.
- For example XML code illustrating message content filtering by using XPath selectors, see "Filtering the message content of publications" on page 1410.
- For an example WSDL document describes a web service that implements the NotificationConsumer portType, see "Example: Creating a Notification consumer web service skeleton" on page 1421.

Writing a WS-Notification application that exposes a web service endpoint

Write a Java EE application, containing a web service definition, that can be deployed to the application server and act as a NotificationProducer, NotificationConsumer or demand-based publisher.

Before you begin

This task assumes that you have the following resources:

- An installed and functioning copy of IBM Rational Application Developer, Rational Software Architect or equivalent tooling.
- The WSDL file for the endpoint that is to be exposed.

About this task

To write a WS-Notification application that exposes a Web service endpoint, follow the method provided by your tooling for creating a web service implementation from a WSDL file. For example in Rational Software Architect there is a wizard in the Tutorials Gallery under "Create and deploy a WS-I compliant web service

and an enterprise bean skeleton from a WSDL document by using the WebSphere Application Server runtime environment". This wizard guides you through the following steps for writing a JAX-RPC application. The steps are very similar for writing a JAX-WS application. For an example of a JAX-WS NotificationConsumer client application that exposes a web service endpoint, see Writing JAX-WS applications for WS-Notification.

Procedure

1. Create a Dynamic Web Project.
2. Import and validate the WSDL file.
3. Create the web service. Select **File** > **New** > **Other** > **Web services** > **Web service wizard** > **Skeleton EJB web service**.
4. Implement the business methods in the generated EJB class. The methods you choose depend upon the type of endpoint that you are exposing (NotificationProducer, NotificationConsumer or demand based publisher).
5. Export the application.

What to do next

You are now ready to deploy the application to WebSphere Application Server as described in Installing enterprise application files with the console. In the Select installation options panel, ensure that the **Deploy web services** option is enabled.

Writing a WS-Notification application that does not expose a web service endpoint

Write a Java EE application that can be run outside of the application server to make web service invocations against an external web service. This application acts as a lightweight publisher, or a pull type consumer by invoking web service operations against another web service such as the NotificationBroker provided by WebSphere Application Server.

Before you begin

This task assumes that you have the following resources:

- An installed and functioning copy of IBM Rational Application Developer, Rational Software Architect or equivalent tooling.
- Knowledge of where to find the WSDL file for the service that is to be invoked.

About this task

To write a WS-Notification application that does not expose a web service endpoint, follow the method provided by your tooling for creating a web service implementation from a WSDL file. As an illustration, the following steps describe the method provided by Rational Software Architect for writing a JAX-RPC application. The steps are very similar for writing a JAX-WS application. For examples of JAX-WS publisher and subscriber client applications that do not expose a web service endpoint, see Writing JAX-WS applications for WS-Notification.

Procedure

1. Get the WSDL files for the service that you want to invoke. If the target service is the notification broker service that was generated by WebSphere Application Server, use the administrative console to publish the WSDL files for the service to a compressed file.
2. Create a Dynamic Web Project with a name of your choice.
3. Choose **File > New > Other > Web services > Web services Client**.
4. Select **Java Proxy**.

5. Enter or select the WSDL you obtained earlier.
6. Choose a Client Type of “Application Client” or “Java” depending upon your requirements.
7. Select your required security configuration.
8. Click **Finish**.
9. Use the generated proxy and stubs to make calls against the remote web service. For detailed coding examples, see “Developing applications that use WS-Notification” on page 1407.

What to do next

You are now ready to deploy the application for use in the Java EE application client container as described in “Running a Java EE client application with launchClient” on page 2015.

Filtering the message content of publications

Rather than receiving all messages on a topic to which you have subscribed, your consuming application can use XML Path (XPath) selectors to filter the messages based upon the contents of each message. This content-based subscription gives you greater flexibility in defining the type of information that you want to receive, and your applications do not need to handle their own filtering. Performance is improved because messages that are not relevant are not sent unnecessarily from the server to the application.

About this task

The WS-Notification publish and subscribe messaging model is topic-based. Each publication is classified as belonging to one of a fixed set of topics. Publishers label each publication with a topic name and consumers subscribe to all publications on a particular topic. For example a stock trading notification system might define a topic for each issue: Publishers post information labeled with the appropriate issue as the topic name, and subscribers subscribe to information regarding some issue.

You can use XPath selectors to filter messages for a given topic, by using a Boolean expression that is evaluated over the XML message content of the message body. For example, a subscriber to a topic-based publish and subscribe system for stock trading might use XPath selectors to specify constraints against three message attributes at the same time:

- issue name
- price
- volume of shares

The resultant Boolean statement might be as follows:

```
(issue="IBM") and (price<120) and (volume>1000)
```

You code your XPath 1.0 message content filters in the subscribing applications, by using XML Path (XPath) language, Version 1.0..

Note: If your subscriber applications use message content filtering, and are coded to specify the XPath Version 1.0 SelectorDomain, they can also filter publications from other WS-Notification providers that are of type JMS TextMessage or BytesMessage. For more information about these JMS message types, see Topology for WS-Notification as an entry or exit point to the service integration bus.

To filter the message content of publications by using XPath selectors, complete the following steps.

Procedure

1. Create a new application that subscribes a WS-Notification consumer.
2. Code an XPath message content filter in the subscribing application. For example code for doing this, see “Example: Subscribing a WS-Notification consumer” on page 1411. For an example of message content filter usage, see the example at the end of this task.

3. Code error handling for cases where the filter is not valid.
4. Invoke the application.

Example

This example XML code illustrates message content filtering by using XPath selectors. In this example a business, represented by a NotificationConsumer application, wants to be notified of bank transfers of over $1,000,000. The monitoring application subscribes on behalf of the NotificationConsumer specifying a valid XPath Version 1.0 message content filter, in the following WS-Notification subscribe message:

```
<wsnt:Subscribe>
  <wsnt:ConsumerReference>
    wsa:EndpointReference
  </wsnt:ConsumerReference>
  <wsnt:Filter>
    [ <wsnt:TopicExpression Dialect="xsd:anyURI">
        {any} ?
      </wsnt:TopicExpression> |
      <wsnt:ProducerProperties Dialect="xsd:anyURI">
        {any} ?
      </wsnt:ProducerProperties> |
      <wsnt:MessageContent Dialect="xsd:anyURI">
        /bankTransfer[value %gt; 1,000,000]
      </wsnt:MessageContent> |
      {any} *
    ] *
  </wsnt:Filter> ?
  <wsnt:InitialTerminationTime>
    [xsd:dateTime | xsd:duration]
  </wsnt:InitialTerminationTime> ?
  <wsnt:SubscriptionPolicy>
    [ <wsnt:UseRaw/> |
      {any}
    ] *
  </wsnt:SubscriptionPolicy> ?
  {any}*
</wsnt:Subscribe>
```

The WS-Notification service stores the subscription and its filter.

Another WS-Notification application then publishes a notification in which the message body contains the following information:

```
<bankTransfer origin="123456 87654321" target="224466 88664422">
  <originName>IBM Corporation</originName>
  <targetName>Matt Roberts</targetName>
  <date>02/02/2006</date>
  <value currency="USD">100,000,000</value>
</bankTransfer>
```

The WS-Notification service in the application server matches this publication to the earlier subscription and delivers the notification to the consumer specified in the subscription.

Example: Subscribing a WS-Notification consumer

Use this task to write the code for a JAX-RPC client acting in the publisher registration role, registering a publisher (producer) application with a broker, based on the example code extract provided.

About this task

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs with code generated by using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

In WebSphere Application Server there are two implementations of the WS-Notification service: Version 6.1 and Version 7.0. This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

The article Writing JAX-WS applications for WS-Notification includes an example of a JAX-WS subscriber client application.

Raw subscriptions:

In this example, the first optional code block shows you how to create a *raw subscription*. This concept is defined in section 4.2 of the Web Services Base Notification specification.

In the normal case, a wrapped subscription causes the Notify operation of the NotificationConsumer to be driven when matching event notifications become available. If the Subscriber instead creates a raw subscription, then only the application specific content of the event notification (that is, the contents of the NotificationMessage element) are sent to the target consumer endpoint. Note that the web service endpoint specified in the ConsumerReference of the Subscribe request that also specifies **UseRaw** (that is, a raw subscription request) does not have to implement the NotificationConsumer port type because the Notify operation will not be used to deliver event notifications.

This means that the consumer must be able to accept operations for each of the types of application content that will be published on the specified topic. This pattern allows WS-Notification to invoke a group of existing web service applications that are not WS-Notification aware, but that want to receive the same information.

JAX-WS supports action-based dispatch, and JAX-WS raw consumer applications must accept the `http://docs.oasis-open.org/wsn/bw-2/NotificationConsumer/Notify` action URI. For more information, see the troubleshooting tip A JAX-WS application that is a raw consumer of brokered notifications must recognize a notification broker SOAP action.

To write the code for a JAX-RPC client acting in the publisher registration role, registering a publisher (producer) application with a broker, complete the following steps, referring to the example code extract for further information.

Procedure

1. Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation.
2. Get a stub for the port on which you want to invoke operations.
3. Create the ConsumerReference. This either contains the address of the consumer web service that is being subscribed to, or a reference to a pull point. Specifying a pull point EPR indicates that the consumer is to use pull-based notifications.
4. Create the filter. This provides the name of the topic to which you want to subscribe the consumer.

5. Create a topic expression and add it to the filter. The prefixMappings are mappings between namespace prefixes and their corresponding namespaces for prefixes used in the expression.
6. Create an XPath 1.0 message content filter. For example you could select a subset of the available messages in the topic, based upon salary level. For an example of message content filter usage, see “Filtering the message content of publications” on page 1410.
7. Create the InitialTerminationTime. This is the time when you want the subscription to terminate. For example, you could set a time of 1 year in the future.
8. Create the Policy information.
9. Optional: Construct a policy indicating that the consumer is to receive raw style notifications.
10. Create holders to hold the multiple values returned from the broker:
    - The subscription reference
    - The current time at the broker
    - The termination time for the subscription
    - Any additional elements
11. Invoke the Subscribe operation by calling the associated method on the stub.
12. Get the returned values:
    - An endpoint reference for the subscription that has been created. This is required for subsequent lifetime management of the subscription, for example pausing the subscription.
    - The current time at the broker.
    - The termination time of the subscription.
    - Any other information.

Example

The following example code describes a subscriber client application that can subscribe a consumer application with a broker:

```
// Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create the ConsumerReference. This contains the address of the consumer web service that is being
// subscribed, or alternatively is a reference to a pull point (see alternative below). Specifying a
// pull point EPR indicates that the consumer is to use pull-based notifications.
EndpointReference consumerEPR =
    EndpointReferenceManager.createEndpointReference(new URI("http://myserver.mycom.com:9080/Consumer"));

/*
// Alternative ConsumerReference for pull-based notifications:

EndpointReference consumerEPR = pullPointEPR;

*/

// Create the Filter. This provides the name of the topic to which you want to subscribe the consumer
Filter filter = new Filter();

// Create a topic expression and add it to the filter. The prefixMappings are mappings between namespace
// prefixes and their corresponding namespaces for prefixes used in the expression
Map prefixMappings = new HashMap();
prefixMappings.put("abc", "uri:example");
TopicExpression exp =
    new TopicExpression(TopicExpression.SIMPLE_TOPIC_EXPRESSION, "abc:ExampleTopic", prefixMappings);
filter.addTopicExpression(exp);


//Create an XPath 1.0 message content filter
//This example selects a subset of the available messages in the topic, based upon salary level
String filterExpression = "/company/department/employee/salary > 10000";
URI xpathURI = new URI(http://www.w3.org/TR/1999/REC-xpath-19991116);
```

```
QueryExpression qexp =
    new QueryExpression(xpathURI, filterExpression);

filter.addMessageContentExpression(qexp);

// Create the InitialTerminationTime. This is the time when you want the subscription to terminate.
// For example, set a time of 1 year in the future.
Calendar cal = Calendar.getInstance();
cal.add(Calendar.YEAR, 1);
AbsoluteOrRelativeTime initialTerminationTime = new AbsoluteOrRelativeTime(cal);

// Create the Policy information
SOAPElement[] policyElements = null;

/*
Optional
--------
The following lines show how to construct a policy indicating that the consumer is to
receive raw style notifications:

    javax.xml.soap.SOAPFactory soapFactory = javax.xml.soap.SOAPFactory.newInstance();
    SOAPElement useRawElement = null;

    if (soapFactory instanceof IBMSOAPFactory) {
        // You can use the value add methods provided by the IBMSOAPFactory API to create the SOAPElement
        // from an XML string.
       String useRawElementXML = "<mno:UseRaw xmlns:mno=\"http://docs.oasis-open.org/wsn/b-2\"/>";
        useRawElement = ((IBMSOAPFactory) soapFactory).createElementFromXMLString(useRawElementXML);
    } else {
        useRawElement = soapFactory.createElement("UseRaw", "mno", "http://docs.oasis-open.org/wsn/b-2");
    }

    policyElements =  new SOAPElement[] { useRawElement };
*/

// Create holders to hold the multiple values returned from the broker:
// The subscription reference
EndpointReferenceTypeHolder subscriptionRefHolder = new EndpointReferenceTypeHolder();

// The current time at the broker
CalendarHolder currentTimeHolder = new CalendarHolder();

// The termination time for the subscription
CalendarHolder terminationTimeHolder = new CalendarHolder();

// Any additional elements
AnyArrayHolder anyOtherElements = new AnyArrayHolder();

// Invoke the Subscribe operation by calling the associated method on the stub
stub.subscribe(consumerEPR,
                       filter,
                       initialTerminationTime,
                       policyElements,
                       anyOtherElements,
                       subscriptionRefHolder,
                       currentTimeHolder,
                       terminationTimeHolder);

// Get the returned values:
// An endpoint reference for the subscription that has been created. It is required for
// subsequent lifetime management of the subscription, for example pausing the subscription
com.ibm.websphere.wsaddressing.EndpointReference subscriptionRef = subscriptionRefHolder.value;

// The current time at the broker
Calendar currentTime = currentTimeHolder.value;

// The termination time of the subscription
Calendar terminationTime = terminationTimeHolder.value;

// Any other information
SOAPElement[] otherElements = anyOtherElements.value;
```

Example: Pausing a WS-Notification subscription

Use this task to write the code for a JAX-RPC client acting in the subscriber role, pausing a subscription for a consumer application, based on the example code extract provided.

About this task

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs with code generated by using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

In WebSphere Application Server there are two implementations of the WS-Notification service: Version 6.1 and Version 7.0. This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

Procedure

1. Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation. The PauseSubscription operation belongs to the SubscriptionManager service.
2. Get a stub for the port on which you want to invoke operations.
3. Associate the request with the subscription you want to pause. The subscriptionEPR is the EndpointReference returned by the invocation of the Subscribe operation.
4. Create any optional information.
5. Invoke the PauseSubscription operation by calling the associated method on the stub.

Example

The following example code describes a JAX-RPC client acting in the subscriber role, pausing a subscription for a consumer application:

```
// Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation.
// The PauseSubscription operation belongs to the SubscriptionManager service
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup("java:comp/env/services/SubscriptionManager");

// Get a stub for the port on which you want to invoke operations
SubscriptionManager stub = (SubscriptionManager) service.getPort(SubscriptionManager.class);

// Associate the request with the subscription you want to pause. The subscriptionEPR is the
// EndpointReference returned by the invocation of the Subscribe operation
((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, subscriptionEPR);

// Create any optional information
SOAPElement[] optionalInformation =  new SOAPElement[] {};

// Invoke the PauseSubscription operation by calling the associated method on the stub
SOAPElement[] additionalReturnedInformation = stub.pauseSubscription(optionalInformation);
```

Example: Publishing a WS-Notification message

Use this task to write the code for a publisher client application that can publish a notification message to a broker, based on the example code extract provided.

About this task

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs with code generated by using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

In WebSphere Application Server there are two implementations of the WS-Notification service: Version 6.1 and Version 7.0. This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

The article Writing JAX-WS applications for WS-Notification includes an example of a JAX-WS publisher client application.

To write the code for a publisher client application that can publish a notification message to a broker, complete the following steps, referring to the example code extract for further information.

Procedure

1. Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation.
2. Get a stub for the port on which you want to invoke operations.
3. Create the message contents for a notification message.
4. Create a notification message from the contents.
5. Add a topic expression to the notification message. The topic expression must indicate to which topic or topics the message corresponds.
6. Create any optional information.
7. Optional: If the broker requires publisher client applications to register, associate the request with a particular publisher registration. The registrationEPR is the ConsumerReference EndpointReference returned by the broker in relation to an invocation of the RegisterPublisher operation.
8. Invoke the Notify operation by calling the associated method on the stub.

Example

The following example code represents a publisher client application that can publish a notification message to a broker:

```
// Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create the message contents for a notification message
SOAPElement messageContents = null;
javax.xml.soap.SOAPFactory soapFactory = javax.xml.soap.SOAPFactory.newInstance();
if (soapFactory instanceof IBMSOAPFactory) {
    // You can use the value add methods provided by the IBMSOAPFactory API to create the SOAPElement
    // from an XML string.
    String messageContentsXML = "<xyz:MyData xmlns:xyz=\"uri:mynamespace\">Some data</xyz:MyData>";
```

```
    messageContents = ((IBMSOAPFactory) soapFactory).createElementFromXMLString(messageContentsXML);
} else {
    // Build up the SOAPElement using the standard javax.xml.soap APIs
    messageContents = soapFactory.createElement("MyData", "xyz", "uri:mynamespace");
    messageContents.addTextNode("Some data");
}

// Create a notification message from the contents
NotificationMessage message = new NotificationMessage(messageContents);

// Add a topic expression to the notification message indicating to which topic or topics the
// message corresponds
Map prefixMappings = new HashMap();
prefixMappings.put("abc", "uri:example");
TopicExpression exp =
    new TopicExpression(TopicExpression.SIMPLE_TOPIC_EXPRESSION, "abc:ExampleTopic", prefixMappings);
message.setTopic(exp);

// Create any optional information
SOAPElement[] optionalInformation =  new SOAPElement[] {};

/*
Optional
--------
The following line will cause the request to be associated with a particular publisher registration.
You must do this if the broker requires publishers to register. The registrationEPR is the
ConsumerReference EndpointReference returned by the broker in relation to an invocation of the
RegisterPublisher operation.

    ((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, consumerReferenceEPR);
*/

// Invoke the Notify operation by calling the associated method on the stub
stub.notify(new NotificationMessage[] { message }, optionalInformation);
```

Example: Creating a WS-Notification pull point

Use this task to write the code for a JAX-RPC subscriber client. This client creates a pull point for use by consumer applications that use pull style notifications.

About this task

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs with code generated by using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

In WebSphere Application Server there are two implementations of the WS-Notification service: Version 6.1 and Version 7.0. This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

To write the code for a JAX-RPC client acting in the subscriber role, creating a pull point for use by a consumer application that is to use pull style notifications, complete the following steps, referring to the example code extract for further information.

Procedure

1. Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation.
2. Get a stub for the port on which you want to invoke operations.
3. Create the request information.
4. Invoke the CreatePullPoint operation by calling the associated method on the stub.
5. Retrieve the reference to the pull point from the response.
6. Retrieve any additional information from the response.

Example

The following example code describes a JAX-RPC client acting in the subscriber role, creating a pull point for use by a consumer application that is to use pull style notifications:

```
// Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create the request information.
SOAPElement[] optionalInformation = null;
CreatePullPoint cpp = new CreatePullPoint(optionalInformation);

// Invoke the CreatePullPoint operation by calling the associated method on the stub
CreatePullPointResponse response = stub.createPullPoint(cpp);

// Retrieve the reference to the pull point from the response
EndpointReference pullPointEPR = response.getPullPoint();

// Retrieve any additional information from the response
SOAPElement[] additionalInformation = response.getElements();
```

Example: Getting messages from a WS-Notification pull point

Use this task to write the code for a JAX-RPC client acting in the pull style consumer role, requesting messages from a pull point, based on the example code extract provided.

About this task

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs with code generated by using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

In WebSphere Application Server there are two implementations of the WS-Notification service: Version 6.1 and Version 7.0. This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS
- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

To write the code for a JAX-RPC client acting in the pull style consumer role, requesting messages from a pull point, complete the following steps, referring to the example code extract for further information.

Procedure

1. Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation.
2. Get a stub for the port on which you want to invoke operations.
3. Associate the request with a pull point. The pullPointEPR is the EndpointReference returned from invoking the CreatePullPoint operation.
4. Specify the number of messages you want to retrieve.
5. Create any optional information.
6. Create the request information.
7. Invoke the GetMessages operation by calling the associated method on the stub.
8. Get the messages returned from the response.

Example

The following example code describes a JAX-RPC client acting in the pull style consumer role, requesting messages from a pull point:

```
// Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Associate the request with a pull point. The pullPointEPR is the EndpointReference returned
// from invoking the CreatePullPoint operation
((Stub) stub)._setProperty(WSAConstants.WSADDRESSING_DESTINATION_EPR, pullPointEPR);

// Specify the number of messages you want to retrieve
Integer numberOfMessages = new Integer(2);

// Create any optional information
SOAPElement[] optionalInformation =  new SOAPElement[] {};

// Create the request information
GetMessages request = new GetMessages(numberOfMessages, optionalInformation);

// Invoke the GetMessages operation by calling the associated method on the stub
GetMessagesResponse response = stub.getMessages(request);

// Get the messages returned from the response
NotificationMessage[] messages = response.getMessages();
```

Example: Registering a WS-Notification publisher

Use this task to write the code for a subscriber client application that can subscribe a consumer application with a broker, based on the example code extract provided.

About this task

This example is based on using the Java API for XML-based remote procedure calls (JAX-RPC) APIs with code generated by using the WSDL2Java tool (run against the Notification Broker WSDL generated as a result of creating your WS-Notification service point) and WebSphere Application Server APIs and SPIs.

In WebSphere Application Server there are two implementations of the WS-Notification service: Version 6.1 and Version 7.0. This JAX-RPC example can interact successfully with Version 6.1 or Version 7.0 WS-Notification service points. However if you want to use WS-Notification with policy sets, for example to enable composition with WS-ReliableMessaging, then your WS-Notification applications must be encoded to use the Java API for XML-based Web Services (JAX-WS) programming model and must interact with Version 7.0 WS-Notification service points. If you are new to programming JAX-WS client applications, see the following topics:

- JAX-WS

- JAX-WS client programming model
- Implementing static JAX-WS web services clients
- Writing JAX-WS applications for WS-Notification
- Web services hints and tips: JAX-RPC versus JAX-WS, Part 1

To write the code for a subscriber client application that can subscribe a consumer application with a broker, complete the following steps, referring to the example code extract for further information.

Procedure

1. Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation.
2. Get a stub for the port on which you want to invoke operations.
3. Create a reference for the publisher (producer) being registered. This contains the address of the producer web service.
4. Create a list (array) of topic expressions to describe the topics to which the producer publishes messages.
5. Indicate that you do not want the publisher to use demand based publishing.
6. Set a value for the initial termination time of the registration. For example, you could set a value 1 year in the future.
7. Create holders to hold the multiple values returned from the broker:
   - **PublisherRegistrationReference** defines the endpoint reference for use in lifetime management of the registration.
   - **ConsumerReference** defines the endpoint reference for use in subsequent publishing of messages.
8. Invoke the RegisterPublisher operation by calling the associated method on the stub.
9. Retrieve the PublisherRegistrationReference.
10. Retrieve the ConsumerReference.

Example

The following example code represents a JAX-RPC client acting in the publisher registration role, registering a publisher (producer) application with a broker.

```
// Look up the JAX-RPC service. The JNDI name is specific to your web services client implementation
InitialContext context = new InitialContext();
javax.xml.rpc.Service service = (javax.xml.rpc.Service) context.lookup(
    "java:comp/env/services/NotificationBroker");

// Get a stub for the port on which you want to invoke operations
NotificationBroker stub = (NotificationBroker) service.getPort(NotificationBroker.class);

// Create a reference for the publisher (producer) being registered. This contains the address of the
// producer web service.
EndpointReference publisherEPR =
    EndpointReferenceManager.createEndpointReference(new URI("http://myserver.mysom.com:9080/Producer"));

// Create a list (array) of topic expressions to describe the topics to which the producer publishes
// messages. For this example you add one topic
Map prefixMappings = new HashMap();
prefixMappings.put("abc", "uri:mytopicns");
TopicExpression topic =
    new TopicExpression(TopicExpression.SIMPLE_TOPIC_EXPRESSION, "abc:xyz", prefixMappings);
TopicExpression[] topics = new TopicExpression[] {topic};

// Indicate that you do not want the publisher to use demand based publishing
Boolean demand = Boolean.FALSE;

// Set a value for the initial termination time of the registration. For example, set a value 1 year in
// the future
Calendar initialTerminationTime = Calendar.getInstance();
initialTerminationTime.add(Calendar.YEAR, 1);

// Create holders to hold the multiple values returned from the broker:
// PublisherRegistrationReference: An endpoint reference for use in lifetime management of
// the registration
```

```
EndpointReferenceTypeHolder pubRegMgrEPR = new EndpointReferenceTypeHolder();

// ConsumerReference: An endpoint reference for use in subsequent publishing of messages
EndpointReferenceTypeHolder consEPR = new EndpointReferenceTypeHolder();

// Invoke the RegisterPublisher operation by calling the associated method on the stub
stub.registerPublisher(publisherEPR, topics, demand, initialTerminationTime, null, pubRegMgrEPR, consEPR);

// Retrieve the PublisherRegistrationReference
EndpointReference registrationEPR = pubRegMgrEPR.value;

// Retrieve the ConsumerReference
EndpointReference consumerReferenceEPR = consEPR.value;
```

Example: Creating a Notification consumer web service skeleton

Use this example when creating a web service that implements the NotificationConsumer portType defined by the Web Services Base Notification specification.

About this task

This task provides two code examples:

- An example WSDL document that describes a web service that implements the NotificationConsumer portType defined by the Web Services Base Notification specification.
- A basic implementation of the Service Endpoint Interface (SEI) generated from the preceding WSDL document using the WSDL2Java tool.

Note: The article Writing JAX-WS applications for WS-Notification also includes an example of a consumer Web service.

Procedure

If you are creating a Notification consumer web service skeleton, see the following code examples.

Example

The following example WSDL document describes a web service that implements the NotificationConsumer portType defined by the Web Services Base Notification specification:

```
<?xml version="1.0" encoding="utf-8"?>

<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:wsn-bw="http://docs.oasis-open.org/wsn/bw-2"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="uri:example.wsn/consumer"
    targetNamespace="uri:example.wsn/consumer">

    <wsdl:import namespace="http://docs.oasis-open.org/wsn/bw-2"
                         location="http://docs.oasis-open.org/wsn/bw-2.wsdl" />

    <wsdl:binding name="NotificationConsumerBinding" type="wsn-bw:NotificationConsumer">
        <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="Notify">
            <wsdlsoap:operation soapAction="" />
            <wsdl:input>
                <wsdlsoap:body use="literal" />
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="NotificationConsumerService">
        <wsdl:port name="NotificationConsumerPort" binding="tns:NotificationConsumerBinding">
            <wsdlsoap:address location="http://myserver.mycom.com:9080/Consumer" />
```

```
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

The following example shows a basic implementation of the Service Endpoint Interface (SEI) generated from the preceding WSDL document using the WSDL2Java tool:

```
public class ConsumerExample implements NotificationConsumer {

    public void notify(NotificationMessage[] notificationMessage, SOAPElement[] any)
                                                                    throws RemoteException {
        // Process each NotificationMessage
        for (int i=0; i<notificationMessage.length; i++) {
            NotificationMessage message = notificationMessage[i];

            // Get the contents of the message
            SOAPElement messageContent = message.getMessageContents();

            // Get the expression indicating which topic the message is associated with
            TopicExpression topic = message.getTopic();

            // Get a reference to the producer (this value is optional and so might be null)
            EndpointReference producerRef = message.getProducerReference();

            // Get a reference to the subscription (this value is optional and so might be null)
            EndpointReference subscriptionRef = message.getSubscriptionReference();

            // User defined processing ...

        }
    }
}
```

Chapter 32. Developing web services - Reliable messaging (WS-ReliableMessaging)

To configure a web service application to use WS-ReliableMessaging, you attach a policy set that contains a WS-ReliableMessaging policy type. This policy type offers a range of qualities of service: managed persistent, managed non-persistent, or unmanaged non-persistent.

Developing a reliable web service application

Develop a Java API for XML-Based Web Services (JAX-WS) provider or requester application that can compose with a WS-ReliableMessaging-enabled policy set.

Before you begin

At any stage - that is, before or after you have built your reliable web service application, or configured your policy sets - you can set a property that configures endpoints to only support clients that use reliable messaging. This setting is reflected by WS-Policy if engaged.

About this task

You can develop a Java API for XML-Based Web Services (JAX-WS) web service requester application that sends messages reliably, or a web service provider application that requires reliable messaging. Your client application can also take programmatic control of WS-ReliableMessaging sequences.

Procedure

1. Develop your JAX-WS web service application.

   For a Web service requester application that sends messages reliably:

   a. Get an implementation WSDL document, and select the SOAP over HTTP binding. The WSDL should be WS-I Basic Profile compliant.

   b. Build the JAX-WS application from the WSDL implementation document.

   c. (Optional) Enable transaction support for outbound (requester) one-way message sends. For more information, see “Providing transactional recoverable messaging through WS-ReliableMessaging” on page 1426.

   d. (Optional) Use the waitUntilSequenceCompleted method on the sequenceManager to ensure that reliable messaging state is released after the client finishes messaging, as described in the “Controlling WS-ReliableMessaging sequences programmatically” on page 1424 topic.

   e. (Optional) If you want to use in-order delivery (that is, you want WS-ReliableMessaging to make messages available to your requester application in the order that they were sent), then you must also configure your requester application to poll for the messages in the order in which it should receive them. For more information, see Configuring the WS-ReliableMessaging policy.

   For a web service provider application that requires reliable messaging:

   a. Write or get an interface WSDL document that describes the service interface. The document should be compliant with the WS-I Basic Profile.

   b. Write or get an implementation WSDL document, and select the SOAP over HTTP binding. The WSDL should remain WS-I Basic Profile compliant.

   c. Build the JAX-WS application from the WSDL implementation document.

2. Enable your client application to take programmatic control of WS-ReliableMessaging sequences.

   This helps manage resources on the server, for example by removing sequences after a client application has finished messaging. You can add code to create sequences, send acknowledgement

requests, close sequences, terminate sequences and wait until sequences are complete. For more information, including example code, see “Controlling WS-ReliableMessaging sequences programmatically.”

What to do next

You are now ready to configure a policy set instance to enable WS-ReliableMessaging.

Controlling WS-ReliableMessaging sequences programmatically

Your client application can use the WSRMSequenceManager, part of the WebSphere Application Server SPI for reliable messaging, to gain programmatic control over reliable messaging sequences. This helps manage resources on the server, for example by removing sequences after a client application has finished messaging. You can add code to create sequences, send acknowledgement requests, close sequences, terminate sequences and wait until sequences are complete.

Before you begin

The WebSphere Application Server SPI for reliable messaging always uses the static policy set configuration that is applied to the client from which the SPI is called. It does not use any alternative policy set that is subsequently configured by WS-Policy to meet the requirements of a WS-Policy intersection.

About this task

By closing sequences programmatically, you limit the number of open sequences a single client has to support in a single JVM at one time.

For your client application to gain programmatic control over reliable messaging sequences, it needs access to a WSRMSequenceManager instance. Use the following code fragment to achieve this:

```
import com.ibm.wsspi.wsrm.WSRMSequenceManager;
import com.ibm.wsspi.wsrm.WSRMSequenceManagerFactory;

.........

     // Get the factory
     WSRMSequenceManagerFactory factory = WSRMSequenceManagerFactory
         .getInstance();

     // Get the sequence manager instance
     WSRMSequenceManager sequenceManager = factory.createWSRMSequenceManager();
```

All WSRMSequenceManager methods take the following parameters:

- The client instance object. This is either a **Dispatch client** instance, or the **Dynamic proxy client**. For details of the client types, see the JAX-WS client programming model topic.
- The **Port QName** instance for the target endpoint.

To control WS-ReliableMessaging sequences programmatically, add code to your client application as described in the following steps:

Procedure

- Add code to create a sequence.

  To set the available properties use the following methods:

```
/**
 * Sets the target provider endpoint.
 * A null value will cause a NullPointerException when the WSRMSequenceProperties object is used.
 *
 * @param providerEndPoint The target service endpoint URI
 */
public void setTargetEndpointUri(String providerEndPoint);

/**
 * This is used to indicate that a response flow is required between the provider and requester and the response
 * flow will be established at create sequence time
```

```
 *
 * By calling this method it will indicate that a response flow is required.
 */
public void setUseOfferedSequenceId();

/**
 * Set the Soap version for RM protocol messages.
 * The default value for this property is WSRMSequenceProperties.SOAP_11
 *
 * @param soapVersion
 */
public void setSoapVersion(int soapVersion);

/**
 * If the Sequence Acknowledgement messages are to be sent back asynchronously call this method.
 *
 */
public void useAsyncTransport();
```

To create the reliable messaging sequence use the createNewWSRMSequence method on the WSRMSequenceManager:

```
/**
 * Initiates a new sequence handshake between this client and the target EPR specified in the
 * WSRMSequenceProperties instance.
 *
 * This sequence will only be valid for the client issuing the createNewWSRMSequence call.
 *
 * When returning from this call, there is no guarantee that the sequence has been established.
 *
 * @throws NullPointerException if the sequenceProperties object is null, or the target EPR is null
 *
 * @param clientObject The JAX-WS Dispatch instance, or the Dynamic Proxy client instance.
 * @param sequencePropeties The properties for creating the reliable messaging sequence
 * @throws WSRMNotEnabledException
 * @throws WSRMSequenceAlreadyExistsException
 */
public void createNewWSRMSequence(Object clientObject, QName portQName, WSRMSequenceProperties sequencePropeties)

throws WSRMNotEnabledException,
       WSRMSequenceAlreadyExistsException;
```

- Add code to send an acknowledgment request.

  To send an acknowledgment request for a WS-ReliableMessaging sequence, use the following method on the WSRMSequenceManager:

```
/**
 * Sending an acknowledgement request sends the ACK requested message to the specified target endPointUri.
 * The target will respond with a range of messages that can be acknowledged for the current reliable messaging
 * sequence.
 *
 * @param clientObject The JAX-WS Dispatch instance, or the Dynamic Proxy client instance.
 * @param portQName
 * @param endPointUri The target endpoint uri
 * @throws WSRMNotEnabledException
 * @throws WSRMSequenceUnknownException
 * @throws WSRMSequenceTerminatedException
 * @throws WSRMSequenceClosedException
 */
public void sendAcknowledgementRequest(Object clientObject, QName portQName, String endPointUri)

throws WSRMNotEnabledException,
       WSRMSequenceUnknownException,
       WSRMSequenceTerminatedException,
       WSRMSequenceClosedException;
```

- Add code to close a sequence.

  To close a WS-ReliableMessaging sequence use the following method on the WSRMSequenceManager:

```
/**
 * Closes the web services reliable messaging session from this application to
 * the endpoint url specified.
 *
 * Throws a WSRMSequenceTerminatedException if the session between this application
 * and the target endpoint url is already closed
 *
 * Throws a WSRMSequenceTerminatedException when the session between this application
 * and the target endpoint has already been terminated.
 *
 * Throws WSRMSequenceUnknownException exception when either reliable messaging is not engaged to
 * the specified endpoint url or the sequence has previously been terminated and removed.
 *
 * @param clientObject The JAX-WS Dispatch instance, or the Dynamic Proxy client instance.
 * @param endPointUri The target endpoint url
 *
 * @throws WSRMNotEnabledException
```

```
   * @throws WSRMSequenceUnknownException
   * @throws WSRMSequenceClosedException
   * @throws WSRMSequenceTerminatedException
   */
  public void closeSequence(Object clientObject, QName portQName, String endPointUri)

  throws WSRMNotEnabledException,
         WSRMSequenceUnknownException,
         WSRMSequenceClosedException,
         WSRMSequenceTerminatedException;
```

- Add code to terminate a sequence.

  To terminate a WS-ReliableMessaging sequence use the following method on the WSRMSequenceManager:

```
  /**
   * Terminates web services reliable messaging session from this application to
   * the endpoint url specified.
   *
   * Throws a WSRMSequenceTerminatedException when the session between this application
   * and the target endpoint has already been terminated.
   *
   * Throws WSRMSequenceUnknownException exception when either reliable messaging is not engaged to
   * the specified endpoint url or the sequence has previously been terminated and removed.
   *
   * @param clientObject The JAX-WS Dispatch instance, or the Dynamic Proxy client instance.
   * @param endPointUri The target endpoint url
   * @throws WSRMNotEnabledException
   *
   * @throws WSRMSequenceTerminatedException
   * @throws WSRMSequenceUnknownException
   */
  public void terminateSequence(Object clientObject, QName portQName, String endPointUri) throws WSRMNotEnabledException;
```

- Add code to wait for a sequence to complete.

  To wait for a reliable messaging sequence to complete, you use a method call that ensures that all messages have been sent and acknowledged by the target service. After the sequence is completed, it is terminated and cleaned up.

  There are two ways of using the waitUntilSequenceCompleted method:

  –

```
public boolean waitUntilSequenceCompleted(Object clientObject,
QName portQName, String endPointUri, long waitTime)
```

    This method call waits for the specified **waitTime** for the reliable messaging sequence to complete. If the sequence does not complete in the specified time, the method returns false. If the sequence does complete in time, the method returns true.

  –

```
public boolean waitUntilSequenceCompleted(Object clientObject,
QName portQName, String endPointUri)
```

    This method call does not return until the reliable messaging sequence is completed.

Providing transactional recoverable messaging through WS-ReliableMessaging

If your WS-ReliableMessaging application runs inside the web container and uses a managed quality of service, you can use WS-ReliableMessaging to provide transactional recoverable messaging.

About this task

The WS-ReliableMessaging transactional model is as follows:

- On the web service requester side, the transaction is between the application and the local managed store.
- The WS-ReliableMessaging protocol delivers the message to the web service provider side, where a different transaction is used between the second managed store and the application being dispatched.

For the outbound (requestor) case on a one-way message send, if the **enableTransactionalOneWay** property is set to `true`, then the send is performed under any transactional context currently held by the application thread. (Note that transactions are not supported under an outbound two-way message exchange).

For the inbound (provider) case, if the **inOrderDelivery** property is set to `true`, then an inbound message is dispatched to the application under a transaction. For an inbound two-way message exchange, the response is also generated under that transaction and is not sent until that transaction has committed.

Note:

WS-ReliableMessaging transactions do not use the WS-AtomicTransactions protocol. The relationship between these two protocols is as follows:

- WS-AtomicTransactions and WS-ReliableMessaging are mutually exclusive when WS-ReliableMessaging is being used, with a managed store, to provide transactional recoverable messaging.
- If WS-ReliableMessaging is configured to use an in-memory store, then there are cases where a WS-AtomicTransaction can be flowed between the reliable messaging source and the reliable messaging destination for two-way invocations. In this situation, WS-ReliableMessaging only protects against network failures, not against server failure.

For more information, see WS-AtomicTransactions.

To provide transactional recoverable messaging through WS-ReliableMessaging, work through the steps described in Adding assured delivery to web services through WS-ReliableMessaging and also complete the following additional steps:

Procedure

- To enable transactional messaging for outbound (requester) one-way message sends, when you develop your JAX-WS web service application set the **enableTransactionalOneWay** property to `Boolean.TRUE` (or the string `true`) in the jaxWS request context map.
- To enable transactional messaging for inbound (provider) one-way and two-way message exchanges, when you configure your WS-ReliableMessaging policy either use the administrative console to select the option **Deliver messages in the order that they were sent** or use the wsadmin tool to set the **inOrderDelivery** property to `true`.

Configuring endpoints to only support clients that use WS-ReliableMessaging

By default, when a WS-ReliableMessaging enabled policy set is attached to an endpoint, the server supports clients that use reliable messaging and clients that do not use reliable messaging. In this version of the product, you can configure endpoints to only support clients that use reliable messaging.

About this task

You configure endpoints to only support clients that use reliable messaging by setting a property in either of the following ways:

- Set a property when packaging the application.
- Set a property as a JVM argument for the server.

This setting is reflected by WS-Policy if engaged. For information about how to engage WS-Policy, see Using WS-Policy to exchange policies in a standard format.

Procedure

- When packaging the application, configure endpoints to only support clients that use reliable messaging by setting the **strictlyEnforceWSRM** property in the META-INF/MANIFEST.MF of a WAR file or EJB module.

- Using a JVM argument for the server, configure endpoints to only support clients that use reliable messaging by defining the Java virtual machine custom property **com.ibm.ws.websvcs.rm.strictlyEnforceWSRM** on the server. For more information, see Configuring the JVM.

Chapter 33. Developing web services - RESTful services

You can use Java API for RESTful Web Services (JAX-RS) to develop services that follow Representational State Transfer (REST) principles. RESTful services are based on manipulating resources. Resources can contain static or dynamically updated data. By identifying the resources in your application, you can make the service more useful and easier to develop.

Planning JAX-RS web applications

Planning to use JAX-RS to enable RESTful services

By using the Java API for RESTful Web Services (JAX-RS) API, application developers can quickly develop RESTful applications. When planning to use JAX-RS to enable RESTful services, consider how to best implement the capabilities and characteristics of a RESTful application with JAX-RS.

Before you begin

Read the overview of JAX-RS information to learn about REST services and the advantages of using JAX-RS to build RESTful services.

About this task

JAX-RS is a programming model that provides a mechanism for developing services that follow Representational State Transfer (REST) principles. Using JAX-RS, development of RESTful services is simplified.

JAX-RS is a Java API for developing REST applications quickly. While JAX-RS provides a faster way for developing web applications than servlets, the primary goal of JAX-RS is to build RESTful services. JAX-RS 1.0 defines a server-side component API to build REST applications. IBM JAX-RS provides an implementation of the JAX-RS (JSR 311) specification.

By using JAX-RS technology, REST applications are simpler to develop, simpler to consume, and simpler to scale when compared to other types of distributed systems. Many popular and widely used Internet services have successfully provided RESTful APIs to their applications. Third parties have used various REST APIs to build their own businesses and applications.

Due to the simple consumption of RESTful services, you can write clients in many languages on different platforms. Most languages require no third-party libraries as long as there is a method to use an HTTP connection. Because of the pervasiveness of web browsers, the most prevalent clients are typically web browsers. For example, many Web 2.0 properties use a JavaScript framework such as Dojo toolkit for developing a client in a browser in conjunction with a RESTful server-side application that provides the data for the client.

Procedure

1. Review existing business and middleware applications in your environment to determine which services you want to implement as REST services.
2. Define the resources in your RESTful applications.
3. Determine the URL patterns, operations, and media type formats to use for each resource.
   a. Define the URI patterns for resources in RESTful applications.
   b. Define the client capabilities for RESTful applications using HTTP methods .
   c. Define the HTTP headers and response codes for RESTful applications using HTTP methods.

Results

You have a design plan for using JAX-RS to implementing REST services.

Defining the resources in RESTful applications

You can use Java API for RESTful Web Services (JAX-RS) to develop services that follow Representational State Transfer (REST) principles. RESTful services are based on manipulating resources. Resources can contain static or dynamically updated data. By identifying the resources in your application, you can make the service more useful and easier to develop.

Before you begin

After you have identified the application that you want to expose as a RESTFUL service, you must first define the resources for your RESTful application. When defining the resources for your application, consider the type of data do you want to expose. Perhaps you already have a relational database that contains information that you want to expose to users using REST technology. Do you already have a set of Java classes defined for accessing that data?

For example, consider the case of an application defined to support a book store. This application currently has a database with several tables that define the various items in the collection of books and the inventory of each book. In this example, there are a number of ways to represent the data in the database in a RESTful application. One approach is to consider each table as an individual resource, so that each of the verbs in the RESTful request maps to the actions that the database supports on that table such as select, insert, update, delete. This example is a simple approach to creating a RESTful application. This approach using the book store example is also used in the documentation that describes defining URL patterns for resources, resource methods, HTTP headers and response codes, media types, and parameters for request representations to resources

In support of this database for the book store application, there might already be existing code that is responsible for accessing the database and retrieving the data from each table. Even though the rows in each of the tables logically represents each resource, the accessor classes are used to define the resources. The implementing JAX-RS applications documentation provides more details on how these classes are incorporated into your JAX-RS application.

Alternately, you might have more static content that does not reside in a database that you want to distribute as resources. Whether it is a collection of documents in various formats or a resource-based facade for other remote systems, using JAX-RS, you can distribute content from multiple sources.

About this task

Resources are the basic building block of a RESTful service. Examples of a resource from an online book store application include a book, an order from a store, and a collection of users.

Resources are addressable by URLs and HTTP methods can perform operations on resources. Resources can have multiple representations using different formats such as XML and JSON. You can use HTTP headers and parameters to pass additional information that is relevant to the request and response.

With JAX-RS, you can annotate existing or new Plain Old Java objects (POJO) with JAX-RS specific annotations. JAX-RS annotated resource classes and the annotated methods are invoked depending on the URI patterns. You can use the annotated resource classes after these resource classes are added to the list of resources returned by the overridden methods in the JAX-RS application class.

Procedure

1. Identify the types of resources in the application.
2. (optional) Identify existing Java classes that you can use as resource classes.

3. Create new Java classes for resources that do not have an existing Java class.

Results

You have defined the content that you want to expose as a collection of resources in your application.

What to do next

Based on the resources that you have defined, read about defining URL patterns for resources, resource methods, HTTP headers and response codes, media types, and parameters for request representations to resources to learn more about additional steps you can take to define the resources for your JAX-RS application.

Defining the URI patterns for resources in RESTful applications

Representational State Transfer (REST) services are based on manipulating resources. Resources for RESTful services are addressable, and URLs are the primary way of achieving addressability in REST.

Before you begin

Identify the resources in the application that you want to expose as a RESTful service.

About this task

URLs are used to specify the location of a resource. Interaction between the server and client is based on issuing HTTP operations to URLs. Defining URL patterns is important because URLs often have a long lifetime so that clients can directly address a resource long after the resource is initially discovered.

URLs are typically used when you enter addresses to web browsers, such as `http://www.ibm.com/` or `http://www.example.com/bookstore/books/ISBN123`. Although URLs are not required to be understandable by users, RESTful services that provide logical URLs in understandable patterns enable client application developers to work efficiently.

RESTful clients use URLs to manipulate resources. Each resource must have its own unique URL. Some URL patterns have a collection path with a unique identifier appended. For example, you can use `http://www.example.com/bookstore/books` as the collection resource URL, `http://www.example.com/bookstore/books/ISBN123` as a unique book resource URL, and you can use `http://www.example.com/bookstore/books/ISBN123/authors` to retrieve a collection resource describing `ISBN123` authors.

The application developer must carefully consider the granularity of URLs because it can affect usage of the application and performance. For example, you can include the author information of a book as part of the book resource or you can define the author information as a unique resource with its own URL that is referenced in the book resource. Depending on the reuse of resources, it might be more efficient to define a separate resource for the author information that is referenced in a hyperlink of the book resource for cases when the author writes a different book.

After an initial URL is given to a client, subsequent related requests are discoverable by parsing the current resource. In the book example, a GET request to `http://www.example.com/bookstore/books/` retrieves a list of book URLs that can include `http://www.example.com/bookstore/books/ISBN123`.

Because systems rely on resources being available, URLs typically have longevity. Because HTTP has built-in status codes for redirection, such as the 301 moved permanently code and the 307 temporarily redirected code, users and clients with caches often reuse previously discovered URLs first. You can additionally consider including a version identifier in the URL pattern, such as `http://www.example.com/bookstore/v2/books/ISBN123`. Like the planning involved to define an interface using Java code, be sure to carefully choose your URL patterns because of expected longevity.

In Java API for RESTful Web Services (JAX-RS), you must add @Path annotations to the Java class files or the Java methods to define the relative URL of the resource. You can use JAX-RS subresource locators and subresource methods to define resources. Use parameters, such as the path parameter or matrix parameter, in the URL to identify the resource.

The value in the @Path annotation defines the relative part of the full URL to your resource. The base URL is derived from the domain, port, application module context root, and any URL pattern mappings in the web.xml file of the application module. For example, if the domain is `www.example.com`, the port is `9060`, the module context root is `example`, the servlet URL pattern is `store/*`, and the value of the @Path annotation is `/bookstore/books`. The full URL is: `http://www.example.com:9060/example/store/bookstore/books`.

Procedure

1. Identify the types of resources in the application. Suppose that you have two types of resources, a BooksCollection and an individual Book object which have the following class definitions:

```
public class BooksCollection {
    public BooksCollection() {
        /* no argument constructor */
    }

}

public class Book {
    public Book(String ISBN) {
        /* This constructor has an argument that will be annotated with a JAX-RS annotation.
         See the JAX-RS specification for information on valid constructors. */
    }
}
```

   As defined in the JAX-RS specification, by default, resource instances are created per request. For the JAX-RS runtime environment to create a resource instance, you must have either a constructor with no argument or a constructor with only JAX-RS annotated parameters present.

2. Add a @javax.ws.rs.Path annotation to each resource class. For each @javax.ws.rs.Path annotation, set the value as the part of the URL after the base URL of the application.

```
/*
 * BooksCollection.java
 * This Java class represents the books collection URL at /bookstore/books.
 */
@javax.ws.rs.Path("/bookstore/books/")
public class BooksCollection {

}
```

   After completing the application, you can use the resource by visiting `http://<host_name>:<port>/<context_root>/<servlet_path>/bookstore/books`. For this URL, specify the context root value as the part of the URL after the context module. Specify the servlet path as any URL patterns in the web.xml file, if it exists.

3. (optional) Determine if a resource needs to use part of the URL as a parameter. If a resource needs to use part of the URL as a parameter, such as an identifier, you can use the @javax.ws.rs.Path annotation with a regular expression. You can then add a @javax.ws.rs.PathParam annotation in either the resource constructor or the resource method.

```
/*
 * Book.java represents individual books.
  */
@javax.ws.rs.Path("/bookstore/books/{bookID}")
public class Book {
   public Book(@javax.ws.rs.PathParam("bookID") String ISBN) {

   }
}
```

   When an HTTP request is made to `http://<host_name>:<port>/<context_root>/<servlet_path>/bookstore/books/ISBN_number`, a Book instance is created with *ISBN_number* passed in to the ISBN parameter of the constructor.

For more information about other possible parameters, read about defining parameters for requests to resources in RESTful applications.

4. Create the javax.ws.rs.core.Application subclass to define to the JAX-RS runtime environment which classes are part of the JAX-RS application. The resource classes are returned in the getClasses() method; for example:

```
public class BookApplication extends javax.ws.rs.core.Application {
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(BooksCollection.class);
        classes.add(Book.class);
        return classes;
    }
}
```

By defining the javax.ws.rs.core.Application subclass, classes returned from its methods are registered to the JAX-RS runtime environment. When configuring the web.xml file, you must specify the javax.ws.rs.core.Application subclass as a parameter to the servlet or filter. For more information, read about configuring the web.xml file for JAX-RS applications.

Results

You have created a URL to identify your resources for your RESTful service. By considering issues with URL patterns early in the application design, the RESTful service increases its usability and value over an extended time.

What to do next

The resource at the defined URL exists. However, the resource does not yet have any methods to handle HTTP method actions such as GET, POST, PUT, or DELETE. See the defining resource methods for RESTful applications to learn more about defining capabilities of resources using supported HTTP methods.

Defining resource methods for RESTful applications

Individual resources can define their capabilities using supported HTTP methods. In Representational State Transfer (REST) services, the supported methods are GET, PUT, DELETE, and POST. All operations are typically conducted by using one of the predefined HTTP methods with a resource.

Before you begin

Understand the predefined HTTP methods and their known attributes. Some HTTP methods are meant to be safe, meaning that issuing the request does not change the resource, or idempotent, meaning that multiple invocations of the operation do not change the result. While HTTP methods are defined to have certain attributes, the service implementation follows the definitions. See the HTTP method definitions information to learn more about the common set of methods for HTTP.

About this task

Clients use HTTP methods to perform certain operations. Unlike other distributed systems where unique interfaces are defined by each system, RESTful systems based on HTTP mainly rely on predefined methods. The four most common methods are GET, PUT, DELETE, and POST. Resources are not required to permit all HTTP methods for all clients.

The HTTP GET method retrieves a resource representation. It is safe and idempotent. Repeated GET requests do not change any resources.

The HTTP PUT method is often used to update resources or to create a new entity at a known URL. When a resource must be updated or created, an HTTP PUT method is issued at the resource URL with the new resource data as the request entity, also known as the message body. The HTTP PUT method is

idempotent so multiple identical PUT requests with the same entity to the same URL yields the same result as if only one PUT request was issued. This method assumes that no other relevant requests were made.

The HTTP DELETE method removes a resource at a given URL. It is also idempotent.

The HTTP POST method is often used when creating a resource in a collection when the final resource URL is not known. For instance, an administrator issues a POST request to a `/users` collection resource that creates a user with a unique ID such as `1234567890`. The unique ID is then used as part of the URL path to describe the new user resource, such as `/users/1234567790`. It is not safe and is not idempotent. In this case, the multiple POST requests to the `/users` collection can continue creating a new unique ID and adding this new ID to the users collection even if the user has the same information.

Because most RESTful services use well-known HTTP methods that provide expected results, you can more easily create clients. RESTful service developers can take advantage of the expected behaviors of HTTP methods. Resource methods can also use parameters, such as path parameters, query parameters, or matrix parameters to identify the resource. Read about defining the use of parameters for request representations to resources to learn more.

(optional) If you have a sub-resource method and a sub-resource locator method that have an `@Path` annotation with the same value in the same resource class, the sub-resource locator is not considered when determining the method to invoke by default. This is in compliance with the JAX-RS specification.

Use the `wink.searchPolicyContinuedSearch` property with a value of `true` to modify this behavior. This results in sub-resource locators being used if no sub-resource methods match the request.

To enable the property, include a properties file in the `WEB-INF` directory of the module that has the `wink.searchPolicyContinuedSearch` property and value specified. In the `web.xml` file of the application module, include an `init-param` element with the `propertiesLocation` value for the `param-name` element. The `param-value` element specifies the location of the properties file, for example, `WEB-INF/my_wink.properties`.

The following example illustrates the `web.xml` file:

```
<servlet>
    ...
    ...
    <init-param>
        <param-name>propertiesLocation</param-name>
        <param-value>/WEB-INF/my_wink.properties</param-value>
    </init-param>
</servlet>
```

The following example illustrates the `WEB-INF/my_wink.properties` properties file:

```
wink.searchPolicyContinuedSearch=true
```

Procedure

1. Identify the types of resources in the application.

   For each resource class, identify an existing method or create a method that you want to invoke for each supported HTTP method. Methods that respond to HTTP requests are also known as resource methods. For each resource method in the resource class, annotate the Java method with a single JAX-RS HTTP method annotation such as @javax.ws.rs.GET, @javax.ws.rs.POST, @javax.ws.rs.DELETE or @javax.ws.rs.PUT. For example, if an HTTP GET method is supported by the BooksCollection class, then you can create and annotate a method like the following snippet:

```
@javax.ws.rs.Path("/bookstore/books/")
public class BooksCollection {
    @javax.ws.rs.GET
    public String getBooksCollection() {
        String str = /* Construct a String representation of the resource. */
        return str;
    }
}
```

When issuing an HTTP GET request to `http://<host_name>:<port>/<context_root>/<servlet_path>/bookstore/books` URL using a web browser or another HTTP client, the previous getBooksCollection() method is invoked.

2. Ensure that the resource supports the required HTTP methods.

   Each resource typically has multiple resource methods; for example:

```
@javax.ws.rs.Path("/bookstore/books/{bookID}")
public class Book {
    /* This is a database key. */
    private String ISBN;

    public Book(@javax.ws.rs.PathParam("bookID") String ISBN) {
        this.ISBN = ISBN;
    }

    @javax.ws.rs.GET
    public String retrieveSpecificBookInformation() {
        /* This code retrieves a book resource based on the ISBN key. */
    }

    @javax.ws.rs.PUT
    public String updateBookInformation(String updatedBookInfo) {
        /* This code updates the book resource based on ISBN key and the entity (message body) sent
         in the request that is stored in updatedBookInfo. */
    }

    @javax.ws.rs.DELETE
    public void removeBook() {
        /* This code deletes a book resource based on ISBN key. */
    }
}
```

   When issuing an HTTP GET request to the `http://<host_name>:<port>/<context_root>/<servlet_path>/bookstore/books/<isbn_number>` URL using a web browser or another HTTP client, the retrieveSpecificBookInformation() method is invoked. Sending an HTTP PUT request to the same URL invokes the updateBookInformation method and any content in the request message body is passed as the value of the `updatedBookInfo` object. Finally, sending an HTTP DELETE request to the same URL invokes the removeBook() method.

   Note: According to the JAX-RS specification, you must not put multiple HTTP method annotations, such as @javax.ws.rs.POST or @javax.ws.rs.PUT on the same resource method Because HTTP methods have uniquely defined semantics, do not use a resource method for multiple HTTP methods.

Results

You have defined valid operations for the resources.

Defining the HTTP headers and response codes for RESTful applications

HTTP headers and status codes are useful to help intermediary and client programs understand information about requests and responses for applications. HTTP headers contain metadata information. HTTP status codes provide status information about the response.

Before you begin

See the HTTP 1.1 specification to become familiar with HTTP headers and HTTP status codes.

About this task

HTTP headers contain metadata information such as security authentication information, the user agent that is used, and cache control metadata. Standard HTTP headers are defined in the HTTP specification; however, you can use custom HTTP headers, if necessary.

You can read HTTP headers from the request and set the headers in the response. There are a set of common request and response headers, but there are also unique request and unique response headers. JAX-RS provides the HttpHeaders injectable interface and the @HeaderParam parameter annotation for reading HTTP headers. If a javax.ws.rs.core.Response object is returned from a resource method, you can set HTTP headers on the response. Also, you can set HTTP headers when an entity is written using the MessageBodyWriter interface.

You can set HTTP response status codes to help client programs understand the response. While responses can contain an error code in XML or other format, client programs can quickly and more easily understand an HTTP response status code. The HTTP specification defines several status codes that are typically understood by clients.

Procedure

- To read a specific request header, add a @javax.ws.rs.HeaderParam annotated parameter.

```
@javax.ws.rs.Path("/bookstore/books/{bookID}")
public class Book {
    @javax.ws.rs.GET
    public String retrieveSpecificBookInformation(@javax.ws.rs.HeaderParam("User-Agent") String theUserAgent) {
        /* The book ID was sent in a HTTP request header with the name "bookID". */
    }
}
```

- To read any request header, use the @javax.ws.rs.core.Context javax.ws.rs.core.HttpHeaders injected object.

```
@javax.ws.rs.Path("/bookstore/books/{bookID}")
public class Book {
    @javax.ws.rs.GET
    public String retrieveSpecificBookInformation(@javax.ws.rs.core.Context HttpHeaders requestHeaders) {
        /* Call methods on "requestHeaders" to get any request header sent by the client. */
        List<String> bookIdValues = requestHeaders.getRequestHeader("User-Agent");
    }
}
```

- To set a response status code or header, return a javax.ws.rs.core.Response object and build the response with the appropriate status code and headers.

```
@javax.ws.rs.Path("/bookstore/books/{bookID}") public class Book {
    @javax.ws.rs.GET public javax.ws.rs.core.Response retrieveSpecificBookInformation() {
        return Response.status(200).header("responseHeaderName", "responseHeaderValue").header("anotherResponseHeaderName", "foo").build();
    }
}
```

Results

You have used HTTP headers to read request headers and set response status codes and headers for JAX-RS web applications.

Defining media types for resources in RESTful applications

Resources are represented by multiple formats. XML, JavaScript Object Notation (JSON), Atom, binary formats such as PNG, JPEG, GIF, plain text, and proprietary formats are used to represent resources. Representational State Transfer (REST) provides the flexibility to represent a single resource in multiple formats.

Before you begin

Define the resources in the JAX-RS web application.

About this task

Resources have representations. A resource representation is the content in the HTTP message that is sent to, or returned from the resource using the URI. Each representation that a resource supports has a corresponding media type. For example, if a resource is going to return content formatted as XML, you can use `application/xml` as the associated media type in the HTTP message.

Depending on the requirements of your application, resources can return representations in a preferred single format or in multiple formats. For example, resources that are accessed using JavaScript clients might prefer JSON representations because JSON is easy to consume.

JAX-RS provides @Consumes and @Produces annotations to declare the media types that are acceptable for a resource method to read and write.

JAX-RS also maps Java types to and from resource representations using entity providers. A `MessageBodyReader` entity provider reads a request entity and deserializes the request entity into a Java type. A `MessageBodyWriter` entity provider serializes from a Java type into a response entity.

Table 148. Standard entity providers and basic Java types. This table includes the standard entity providers and basic Java types that are included in the JAX-RS runtime environment, along with the corresponding supported content types.

Java type	MessageBodyReader	MessageBodyWriter	Supported Content types
byte[]	X	X	*/*
java.io.InputStream	X	X	*/*
java.io.Reader	X	X	*/*
java.lang.String	X	X	*/*
java.io.File	X	X	*/*
javax.activation.DataSource	X	X	*/*
javax.xml.transform.Source	X	X	text/xml, application/xml, application/*+xml
javax.ws.rs.core.MultivaluedMap	X	X	application/x-www-form-urlencoded
JAXB types	X	X	text/xml, application/xml, application/*+xml
javax.ws.rs.core.StreamingOutput		X	*/*

If a `String` value is used as the request entity parameter, the `MessageBodyReader` entity provider deserializes the request body into a new `String`. If a JAXB type is used as the return type on a resource method, the MessageBodyWriter serializes the Java Architecture for XML Binding (JAXB) object into a response body.

If you need a custom mapping from a Java type to a specific representation, see the information for using an application-defined entity provider.

If your client can handle multiple formats and you want the server to determine the best resource representation to return, read about using content negotiation in JAX-RS applications to serve multiple content types.

The specifications for XML, JSON, and Atom provide details regarding the formats of resource representations for applications. See the specifications to learn more about the formats of resource representations.

Procedure

1. Determine the resource representation format such as XML, JSON, or ATOM to use for either the request or the response.
2. Add the @Consumes and @Produces annotations appropriately to the resource method.
3. If the resource needs to read the content of the request, add a request entity parameter to the resource method. The request entity parameter is a single Java parameter on the method that does not have an annotation.

4. If the resource method returns content in the response, return a Java object that is writable by a JAX-RS entity provider. This Java object is mapped to the response entity in the HTTP response. The returned object must be a JAX-RS supported Java type or wrapped in a javax.ws.rs.core.Response or javax.ws.rs.core.GenericEntity type.

Results

You have mapped the request entities to the resource method entity parameter and any response objects that are returned are mapped to the response entity for the resource representation.

Example

The following example illustrates defining XML as the resource representation for a RESTful bookstore application.

1. Identify the resource methods that you want to read the request entity or return a response entity.

   In the retrieveSpecificBookInformation method example that follows, there is no request entity that is read. However, there is a response object that is returned. This object wraps a JAXB object that contains the entity information. Adding the @Produces annotation on the resource method with a media type of `application/xml` indicates that the resource method always returns an XML representation with a media type of `application/xml`.

   Clients that have an Accept HTTP request header value compatible with the `application/xml` media type invoke the method correctly.

   Clients that do not have an Accept HTTP header value compatible with the `application/xml` media type automatically receive a `406 Not Acceptable` response status which indicates that the server cannot produce an acceptable response.

   The following example identifies the resource methods that read the request entity or return a response entity:

```
/*
 * Book.java
 * This class represents individual books. The @Produces annotation specifies a media type of application/xml.
 */
@Path("/bookstore/books/{bookID}")
public class Book {
  @GET
  @Produces("application/xml")
  public javax.ws.rs.core.Response retrieveSpecificBookInformation(@PathParam("bookID") String theBookID,
   @Context javax.ws.rs.core.HttpHeaders headers) {
   /* ... */
   return
    Response.ok(/* JAXB object to represent response body entity */).expires(/* Expires response header value*/).header("CustomHeaderName", "CustomHeaderValue").build();
   }
   @PUT
   public String updateBookInformation(@PathParam("bookID") String theBookID, String theRequestEntity,
    @javax.ws.rs.HeaderParam("Content-Length") String contentLengthHeader) { /* ... */ }

   @DELETE
   public void removeBook(@PathParam("bookID") String theBookID) { /* ... */ }
}
```

2. Identify the resource methods that need to consume the request information.

   In the following snippet, the PUT method on the book resource accepts the request entity content if a media type of `text/plain` is sent, as defined in the @Consumes annotation. This method returns content with a `text/plain` representation as specified in the @Produces annotations.

   If a client does not send a message with a Content-Type value of `text/plain`, then the PUT resource method is not invoked. If `Content-Type: application/xml` is sent in the HTTP request headers, the `updateBookInformation` Java method is not be called.

   The DELETE method neither reads a request entity nor returns a response entity; therefore, it does not require either an @Consumes or an @Produces annotation.

   The following example identifies the resource methods that consume the request information:

```
/*
 * Book.java
 * This class represents represent individual books with custom headers.
 */
@Path("/bookstore/books/{bookID}")
public class Book {
  @GET
  @Produces("application/xml")
  public javax.ws.rs.core.Response retrieveSpecificBookInformation(@PathParam("bookID") String theBookID, @Context javax.ws.rs.core.HttpHeaders headers) {
```

```
    /* ... */
    return Response.ok(/* JAXB object to represent response body entity */).expires(/* Expires response header value).header("CustomHeaderName", "CustomHeaderValue").build();
  }
  @PUT
  @Consumes("text/plain")
  @Produces("text/plain")
  public String updateBookInformation(@PathParam("bookID") String theBookID, String theRequestEntity, @javax.ws.rs.HeaderParam("Content-Length") String contentLengthHeader) {
    /* ... */
    String responseEntity = /* a plain text representation */;
    return responseEntity;
  }

  @DELETE
  public void removeBook(@PathParam("bookID") String theBookID) { /* ... */ }
}
```

What to do next

See the JAX-RS specification for a list of all the standard media formats that are supported for representations.

Advanced users might consider defining custom mappings of Java types to representations or using content negotiation for clients to negotiate preferred resource representations. To learn more about these options, see the using custom defined entity formats information or the serving multiple content types with content negotiation information.

Defining parameters for request representations to resources in RESTful applications

Parameters are used to pass and add additional information to a request. You can use parameters as part of the URL or in the headers. Path parameters, matrix parameters, query parameters, header parameters, and cookie parameters are useful for passing in additional information to a request.

About this task

Multiple parameters types exist. Java API for RESTful Web Services (JAX-RS) enables easy access to all the types of parameters using annotated injected parameters.

You can use any basic Java primitive type including java.lang.String as parameters, as well as Java types with a constructor that uses a single String or a valueOf(String) static method. Additionally, you can use `List`, `SortedSet`, and `Set` interfaces where the generic type is one of the previously mentioned types, such as a Set when a parameter can have multiple values. If you need to parse requests, then use String as the parameter type to enable you to complete basic inspection and customization of error path responses.

JAX-RS provides the following annotations to use on resource method parameters to specify that the resource method can be invoked with correct parameter values.

javax.ws.rs.PathParam annotation

Path parameters are part of the URL. For example, the URL can include `/collection/{item}`, where `{item}` is a path parameter that identifies the item in the collection. Because path parameters are part of the URL, they are essential in identifying the request.

If parts of the URL are parameters, you can use a @javax.ws.rs.PathParam annotated parameter; for example:

```
@javax.ws.rs.Path("/bookstore/books/{bookId}")
public class BooksCollection {
    @javax.ws.rs.GET
    public String getSpecificBookInfo(@javax.ws.rs.PathParam("bookId") String theBookId) {
        /* theBookId would contain the next path segment after /bookstore/books/  */
    }
}
```

In this example, requests to `/bookstore/books/12345` assigns the value of `12345` to the `theBookId` variable.

javax.ws.rs.MatrixParam annotation

Matrix parameters are part of the URL. For example, if the URL includes the path segment, `/collection;itemID=itemIDValue`, the matrix parameter name is `itemID` and `itemIDValue` is the value.

You can read matrix parameters with a @javax.ws.rs.MatrixParam annotated parameter; for example:

```
@javax.ws.rs.Path("/bookstore/books")
public class BooksCollection {
    @javax.ws.rs.GET
    public String getBookCollectionInfo(@javax.ws.rs.MatrixParam("page") int page, @javax.ws.rs.MatrixParam("filter") String filter) {
        /* This statement uses the page and filter parameters. */
    }
}
```

In this example, requests to`/bookstore/books;page=25;filter=test` invoke the `getBookCollectionInfo` parameter so that the value for the `page` variable is set to `25` and the value for `filter` variable is set to `test`.

javax.ws.rs.QueryParam annotation

Query parameters are appended to the URL after a "?" with name-value pairs. For instance, if the URL is `http://example.com/collection?itemID=itemIDValue`, the query parameter name is `itemID` and `itemIDValue` is the value. Query parameters are often used when filtering or paging through HTTP GET requests.

You can read query parameters with a @javax.ws.rs.QueryParam annotated parameter; for example:

```
@javax.ws.rs.Path("/bookstore/books")
public class BooksCollection {
    @javax.ws.rs.GET
    public String getBookCollectionInfo(@javax.ws.rs.QueryParam("page") int page, @javax.ws.rs.QueryParam("filter") String filter) {
        /* This statement uses the page and filter parameters. */
    }
}
```

In this example, requests to`/bookstore/books;page=25;filter=test` invoke the `getBookCollectionInfo` parameter so that the value for the `page` variable is set to `25` and the value for `filter` variable is set to `test`.

javax.ws.rs.HeaderParam annotation

Header parameters are HTTP headers. While there are pre-defined HTTP headers, you can also use custom headers. Headers often contain control metadata information for the client, intermediary, or server.

If a HTTP request header must be read, use the @javax.ws.rs.HeaderParam annotation; for example:

```
@javax.ws.rs.Path("/bookstore/books/")
public class BooksCollection {
    @javax.ws.rs.GET
    public String getBookCollectionInfo(@javax.ws.rs.HeaderParam("Range") String rangeValue) {
        /* The rangeValue variable contains the value of the HTTP request header "Range" */
    }
}
```

In this example, requests to `/bookstore/books/` with a `Range` HTTP request header value of `bytes=0-499` invokes the method with `bytes=0-499` as the value for the `rangeValue` variable.

javax.ws.rs.CookieParam annotation

Cookie parameters are special HTTP headers. While cookies are associated with storing session identification or stateful data that is not accepted as RESTful, cookies can contain stateless information.

If an HTTP cookie is sent, such as `mycustomid=customvalue123`, you can retrieve the value of the `mycustomid` variable using the following example:

```
@javax.ws.rs.Path("/bookstore/books/")
public class BooksCollection {
    @javax.ws.rs.GET
```

```
    public String getBookCollectionInfo(@javax.ws.rs.CookieParam("mycustomid") String mycustomid) {
        /* The cookie value is passed to the mycustomid variable. */
    }
}
```

javax.ws.rs.FormParam annotation

Form parameters are used when submitting a HTML form from a browser with a media type of `application/x-www-form-urlencoded`. The form parameters and values are encoded in the request message body in the form like the following: `firstParameter=firstValue &secondParameter=secondValue`. The javax.ws.rs.FormParam annotation enables easy access to individual form parameter values.

If a form is submitted and the entity value is `firstName=Bob&lastName=Smith`, you can retrieve the values of the form parameters using the following example:

```
@javax.ws.rs.Path("/customer")
public class Custommer {
    @javax.ws.rs.POST
    public String postCustomerInfo(@javax.ws.rs.FormParam("firstName") String firstName, @javax.ws.rs.FormParam("lastName") String lastName) {
        /* firstName would be "Bob" and secondName would be "Smith" */
    }
}
```

Note: You can either use a single unannotated parameter to represent the message body or use multiple FormParam annotated parameters, but not both. Because the FormParam requires the request message body to be read and the message body is represented as a byte stream, the message body cannot be read again. The following code is not valid:

```
@javax.ws.rs.Path("/bookstore/books")
public class BooksCollection {
    @javax.ws.rs.POST
    public String postSpecificBookInfo(@javax.ws.rs.FormParam("bookId") String theBookId, String theRequestEntity) {
        /* This code is invalid.  Can only use FormParam or a request entity parameter like "String theRequestEntity" and not both */
    }
}
```

Choose one of the following ways to define variables to read parameters.

Procedure

- Add a parameter to the resource method by using an appropriate JAX-RS parameter annotation on the method to identify the type of parameter. You can read multiple types of parameters from a request; for example:

```
@javax.ws.rs.Path("/bookstore/books/")
public class BooksCollection {
    @javax.ws.rs.GET
    public String getBookCollectionInfo(@javax.ws.rs.QueryParam("page") int page, @javax.ws.rs.QueryParam("filter") String filter) {
        /* This statement uses the page and filter parameters. */
    }
}
```

  Issuing an HTTP GET request using a web browser or other HTTP client, such as `http://<host_name>:<port>/<context_root>/<servlet_path>/bookstore/books?page=10&filter=FilterValue`, invokes the getBookCollectionInfo() method with the page set to 10 and filter set to `FilterValue`.

- Add the parameter annotation on the fields, JavaBeans properties, and constructor arguments; for example:

```
@javax.ws.rs.Path("/bookstore/books/")
public class BooksCollection {
    @javax.ws.rs.QueryParam("page") int page;

    String filter;

    @javax.ws.rs.QueryParam("filter")
    public void setFilter(String filter) {
        this.filter = filter;
    }

    @javax.ws.rs.GET
    public String getBookCollectionInfo() {
        /* This statement uses the page and filter parameters. */
    }
}
```

Issuing an HTTP GET request using a web browser or other HTTP client, such as `http://<host_name>:<port>/<context_root>/<servlet_path>/bookstore/books?page=10&filter=FilterValue`, also invokes the getBookCollectionInfo() method with the page set to 10 and filter set to `FilterValue`.

Results

Your resource methods are defined so that they can be invoked with appropriate parameter values.

Example

Defining exception mappers for resource exceptions and errors

Java API for RESTful Web Services (JAX-RS) applications can produce exceptions and errors. The default behavior is to use the exception handling functionality of application container such as JavaServer Pages (JSP) error pages. However, you can customize the error handling and send specific responses back when an exception or error occurs.

About this task

JAX-RS resource methods, like any Java method, can throw checked and unchecked exceptions. By default, an unchecked runtime exception or error occurs in the container again. A checked exception is wrapped in a `ServletException` for resources running in the web container. Therefore, a developer can use error handling facilities such as JSP error pages to handle exceptions thrown from a JAX-RS application.

JAX-RS introduced the exception, `javax.ws.rs.WebApplicationException`. A developer can specify a specific error class name or `javax.ws.rs.core.Response` object when creating a `WebApplicationException`. When the `WebApplicationException` is thrown, the information included in the exception by way of a status class name or Response object is used to serialize a response.

If you cannot throw the exception, `WebApplicationException`, in your code and you cannot use the error handling facilities in the web container, but you want to use a custom error response, then you can create a customized JAX-RS `javax.ws.rs.ext.ExceptionMapper` class to map exceptions to HTTP error responses.

The following procedure illustrates how to write a custom `ExceptionMapper` class.

Procedure

1. Create a class that implements the `javax.ws.rs.ext.ExceptionMapper` class and annotate the class with the `javax.ws.rs.ext.Provider` annotation. This step assumes that your JAX-RS resource can throw the exception, `MyCustomException`, in its methods. The following example illustrates a simple `ExceptionMapper` class:

```
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class CustomExceptionMapper implements ExceptionMapper<MyCustomException> {

    public Response toResponse(MyCustomException exception) {
        return null;
    }

}
```

2. In the `toResponse(MyCustomException)` method, return a `Response` object that contains the customized error response. The following example illustrates a customized `ExceptionMapper.toResponse(MyCustomException)` method:

```
@Provider
public class CustomExceptionMapper implements ExceptionMapper<MyCustomException> {

    public Response toResponse(MyCustomException exception) {
        return Response.status(500).entity("Unfortunately, the application cannot
```

```
            process your request at this time.").type("text/plain").build();
    }

}
```

You can have additional code where you log an error, inspect the exception thrown, or use more complex logic.

3. Package the compiled custom `ExceptionMapper` class with your web application project. If you rely on the annotation scanning capabilities to find all of your JAX-RS classes in your web application, no additional steps are required. However, if you return all of the relevant JAX-RS resource classes and providers in a JAX-RS application subclass method, then you must also add the custom `ExceptionMapper` class to the returned set. The following example illustrates a preexisting `javax.ws.rs.core.Application` subclass:

```
import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.core.Application;

public class MyApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(CustomExceptionMapper.class);
        /* add your additional JAX-RS classes here */
        return classes;
    }
}
```

When exceptions occur in your JAX-RS resource methods, you can customize the HTTP error response so that a user cannot see a stack trace or potentially confidential data. Use an `ExceptionMapper` or the exception handling functionality in the web container to give more helpful responses if the application is not behaving correctly.

Results

You have written a custom `ExceptionMapper` to handle exceptions in your JAX-RS web application.

Developing JAX-RS web applications

Getting started with IBM JAX-RS

JAX-RS is a collection of interfaces and Java annotations that simplifies development of server-side REST applications. By using JAX-RS technology, Representational State Transfer (REST) applications are easier to develop and easier to consume when compared to other types of distributed systems.

About this task

JAX-RS is a Java API for developing REST applications quickly. While JAX-RS provides a faster way of developing web applications than servlets, the primary goal of JAX-RS is to build RESTful services. JAX-RS 1.0 defines a server-side component API to build REST applications. The IBM implementation of JAX-RS provides an implementation of the JAX-RS (JSR 311) specification.

Use this Getting Started guide to help you quickly develop and deploy a simple JAX-RS web application.

This procedure illustrates developing a simple Hello World service that is packaged inside a web application archive (WAR) module.

Procedure

1. Create a Java class. This class is used to represent a type of resource.

```
package com.ibm.jaxrs.sample;

public class HelloWorldResource {

}
```

2. Annotate the Java class with a `javax.ws.rs.Path` annotation. The value of the annotation is the relative part of the URL after the application context. The application context is fully defined during deployment. In JAX-RS terminology, this class is known as a root resource.

```
package com.ibm.jaxrs.sample;

@javax.ws.rs.Path("/helloworld")
public class HelloWorldResource {

}
```

3. Create a Java method that returns a `Hello World!` response. It is intended for the method to be invoked when an HTTP request is received.

```
package com.ibm.jaxrs.sample;

@javax.ws.rs.Path("/helloworld")
public class HelloWorldResource {

    public String sayHelloWorld() {
        return "Hello World!";
    }
}
```

4. Add a `javax.ws.rs.GET` annotation to the Java method.

   Now, whenever an HTTP GET request is received by the application to the `/helloworld` path, the `sayHelloWorld` Java method is invoked. The response message body will contain `Hello World!` as its content.

```
package com.ibm.jaxrs.sample;

@javax.ws.rs.Path("/helloworld")
public class HelloWorldResource {

    @javax.ws.rs.GET
    public String sayHelloWorld() {
        return "Hello World!";
    }
}
```

   The resource implementation is now complete.

5. You must create the JAX-RS `javax.ws.rs.core.Application` configuration subclass. This subclass needs to return the set of Java classes that is relevant to the JAX-RS runtime environment.

```
package com.ibm.jaxrs.sample;

public class HelloWorldAppConfig extends javax.ws.rs.core.Application {
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(com.ibm.jaxrs.sample.HelloWorldResource.class);
        return classes;
    }
}
```

6. Create the `web.xml` web module configuration file.

   The file tells the web container that the web module contains the IBM JAX-RS REST servlet. You must initialize the IBM JAX-RS Rest servlet with the application configuration class.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4" xmlns=http://java.sun.com/xml/ns/j2ee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <servlet-name>HelloWorldApp</servlet-name>
        <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.ibm.jaxrs.sample.HelloWorldAppConfig</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldApp</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

See the servlet mapping that is specified in the previous example. The pattern is used to help determine the final URL of the RESTful service.

7. Add the `j2ee.jar` file to the classpath.

   Get the `j2ee.jar` file from the `app_server_root/dev/JavaEE` directory and add the JAR file to the classpath. Compile the classes.

8. Assemble the web application.

   By using the **jar** command-line tool that is included with the JDK, run the following command:

```
jar cvf helloworld-jaxrs.war *
```

   This command creates a WAR file.

9. Deploy the application onto the application server.

   When deploying the application, you might be prompted to provide a value for the context root of the module. The context root is used to define the application context.

   When using the IBM JAX-RS servlet, the following URL defines the application context:

```
http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>/servlet_mapping_pattern
```

   Root resource URLs that are specified by the `@javax.ws.rs.Path` values are relative to the application context root. Therefore, if the context root is defined as `myapplication` during deployment, the URL pattern is defined in the `web.xml` file as `/rest/*`, and the Java root resource class has a `@javax.ws.rs.Path` value of `/helloworld`. An example of the final URL is:

```
http://localhost:9080/myapplication/rest/helloworld
```

   Now, you can send a client request to the final URL by using a web browser or any other HTTP client.

Results

You have developed and deployed a JAX-RS web application on the application server.

Setting up a development environment for JAX-RS applications

The application server provides command-line tools to develop web services clients and implementations that are based on the Java API for RESTful Web Services (JAX-RS) specification. You must set up your development environment before you start developing web services.

Before you begin

Before you can set up a web services development environment within WebSphere Application Server, you must install WebSphere Application Server. For detailed information about installing the application server, read about installing your application server environment.

To develop JAX-RS applications, the JAX-RS libraries must be added to the class path definition. See the information for your assembly tools to understand how to include libraries on the class path for the JAX-RS application.

About this task

Set up a web services development environment by completing the following actions.

Procedure

1. Set up the environment.

   Run the **setupCmdLine** script from the `. /profile_root/<application_server>/bin` directory.

2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:

```
set PATH=%WAS_PATH%;%PATH%
```

Results

You have set up an environment so that you can develop RESTful web services.

What to do next

Develop your JAX-RS application and configure the `web.xml` file for the JAX-RS servlet.

Development and assembly tools

You can use an Integrated Development Environment to develop, assemble, and deploy Java Platform, Enterprise Edition (Java EE) modules for WebSphere Application Server.

The IBM Rational Application Developer for WebSphere Software product and the IBM WebSphere Application Server Developer Tools for Eclipse product are supported tools for integrated development environments.

This information center refers to the products as the *assembly tools*. However, you can use the products to do more than assemble modules. Use these tools in an integrated development environment to develop, assemble, and deploy Java EE modules.

The Rational Application Developer for WebSphere Software is a more extensive set of tools supporting enterprise development. This workbench has integrated support for WebSphere Application Server Version 6.1 and later. This workbench also supports both the OSGi and Java EE programming models, and contains wizards and visual editors to help you develop Web 2.0, Service Component Architecture (SCA), Java, and Java EE applications. This product contains code quality tools to help you analyze code and improve performance. This product integrates with Rational Team Concert to provide a team-based environment to help developers share information and work collaboratively. The Trial download for Rational Application Developer is available at `http://www.ibm.com/developerworks/downloads/r/rad/`.

IBM WebSphere Application Server Developer Tools for Eclipse is a lightweight set of tools for developing, assembling, and deploying Java EE applications to WebSphere Application Server Version 7.0 and 8.x. This workbench integrates with the application server to help you to quickly deploy and test applications. This product contains wizards and visual editors that support the Java EE programming model.

For documentation on the tools, see “Rational Application Developer documentation.” Topics on application assembly in this information center supplement that documentation.

Important: The assembly tools run on Windows and Linux Intel platforms. Users of WebSphere Application Server on all platforms must assemble their modules using an assembly tool installed on Windows or Linux Intel platforms. To install an assembly tool, follow instructions available with the tool.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root

Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
Refers to the installation root directory for Web Server Plug-ins.

profile_root
Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Configuring JAX-RS web applications

Configuring JAX-RS applications using JAX-RS 1.1 methods

You can configure Java API for RESTful Web Services (JAX-RS) applications in multiple ways depending on your needs. To take advantage of the Java Platform, Enterprise Edition (Java EE) 6 functionality, you can use the annotation scanning capabilities. By using annotation scanning, you can omit a JAX-RS javax.ws.rs.core.Application subclass or have a minimally defined javax.ws.rs.core.Application subclass.

About this task

The JAX-RS 1.1 specification supports several new ways to configure a JAX-RS application. You can use the built-in annotation scanning to help automatically configure the application. You can optionally add javax.ws.rs.core.Application subclasses to your application and then add the URL patterns required using either the javax.ws.rs.ApplicationPath annotation or a web.xml servlet definition. When using the IBM JAX-RS implementation, you do not have to specify the servlet class implementation because it is automatically added to the configuration of the web module by the time the JAX-RS application is started.

When using a web.xml file, you must use a Java Servlet 3.0 web.xml file.

Procedure

- Configure the JAX-RS application with only one JAX-RS default application in the web.xml file. Use this method if you only need one JAX-RS application and all resource classes are located after a single URL pattern. You can also specify security constraints with this method, if needed.
  1. Add all of your JAX-RS resource and provider classes to the WEB-INF/classes or WEB-INF/lib directory for your web application. You do not need to add a javax.ws.rs.core.Application subclass to your web application.
  2. In your web.xml file, add a servlet definition with javax.ws.rs.core.Application as the servlet name. You do not need to add a servlet-class. You must add a servlet URL pattern to the web.xml file.

The application server runtime environment adds the specific IBM JAX-RS implementation to the configuration of the web module by the time the JAX-RS application is started; for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
 version="3.0">
 <servlet>
  <servlet-name>javax.ws.rs.core.Application</servlet-name>
 </servlet>
 <servlet-mapping>
  <servlet-name>javax.ws.rs.core.Application</servlet-name>
  <url-pattern>/rest/*</url-pattern>
 </servlet-mapping>
</web-app>
```

The JAX-RS resources are now available at a URL such as:

http://{*hostname*}:{*port*}/{context_root_of_Web_ module}/{*value_of_Web.xml_URL_pattern*}/{*value_of_@javax.ws.rs.Path*}

- Configure the JAX-RS application using the javax.ws.rs.core.Application subclass and the web.xml file. Use this method if you need multiple JAX-RS applications or require only specific resources in certain JAX-RS applications with specific URL patterns. You can also specify security constraints with this method, if needed.
  1. Create a javax.ws.rs.core.Application subclass. In your javax.ws.rs.core.Application subclass getClasses() or getSingletons() methods, return the relevant JAX-RS resources and providers. If you return empty sets in both the getClasses() and getSingletons() methods, all the JAX-RS resource and provider classes that are found in the application are added to the JAX-RS application subclass; for example:

```
package com.example;

public class MyApplication extends javax.ws.rs.core.Application {

}
```

     This example uses the default implementations of javax.ws.rs.core.Application subclass getClasses() and getSingletons() methods that return empty sets. Therefore, all relevant JAX-RS classes are assumed to be returned by the javax.ws.rs.core.Application subclass.
  2. Add the javax.ws.rs.core.Application subclass to your web application.
  3. Add a partial servlet definition in the web.xml file. The servlet name is the full name of the javax.ws.rs.core.Application subclass. Do not define the servlet-class. You must add a servlet URL pattern to the web.xml file. The application server runtime environment adds the specific IBM JAX-RS implementation to the configuration of the web module by the time the JAX-RS application is started; for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
 version="3.0">
 <servlet>
  <servlet-name>com.example.MyApplication</servlet-name>
 </servlet>
 <servlet-mapping>
  <servlet-name>com.example.MyApplication</servlet-name>
  <url-pattern>/rest/*</url-pattern>
 </servlet-mapping>
</web-app>
```

     The JAX-RS resources are now available at a URL such as:

http://{*hostname*}:{*port*}/{context_root_of_Web_ module}/{*value_of_Web.xml_URL_pattern_for_Application_subclass*}/{*value_of_@javax.ws.rs.Path*}

- Configure the JAX-RS application without a web.xml file. Use this method if you do not want to use a web.xml file. When you use this method, you cannot specify security constraints. To specify security constraints, you must use a web.xml file.

1. Create a javax.ws.rs.core.Application subclass. In your javax.ws.rs.core.Application subclass getClasses() or getSingletons() methods, return the relevant JAX-RS resources and providers. If you return empty sets in both the getClasses() and getSingletons() methods, all the JAX-RS resource and provider classes that are found in the application are added to the JAX-RS application subclass; for example:

```
package com.example;

public class MyApplication extends javax.ws.rs.core.Application {

}
```

   This example uses the default implementations of javax.ws.rs.core.Application subclass getClasses() and getSingletons() methods that return empty sets. Therefore, all relevant JAX-RS classes are assumed to be returned by the javax.ws.rs.core.Application subclass.
2. Add a javax.ws.rs.ApplicationPath annotation to the javax.ws.rs.core.Application subclass. The ApplicationPath annotation is supported with the JAX-RS 1.1 specification. The value of the ApplicationPath annotation is used as the servlet URL pattern which is equivalent to the servlet URL pattern in the web.xml file; for example:

```
package com.example;

@javax.ws.rs.ApplicationPath("rest")
public class MyApplication extends javax.ws.rs.core.Application {

}
```

3. Add the javax.ws.rs.core.Application subclass to your web application. When the application is started, the resources are available at the following URL:

```
http://{hostname}:{port}/{context_root_of_Web_ module}/{value_of_@javax.ws.rs.ApplicationPath}}/{value_of_@javax.ws.rs.Path}
```

Results

You have configured your JAX-RS application using JAX-RS 1.1 supported methods by taking advantage of annotation scanning to help automatically configure the application.

Configuring the web.xml file for JAX-RS servlets

The `web.xml` file contains information about the structure and external dependencies of web components in the module and describes how the components are used at run time. To enable the web container to run Java API for RESTful Web Services (JAX-RS) applications, you can configure the `web.xml` file to point directly to the IBM JAX-RS servlet. When using servlets, you can define a servlet path in the `web.xml` file that is appended to the base URL.

About this task

You can configure the `web.xml` file for your web application to enable the JAX-RS application code. You can specify an IBM specific JAX-RS servlet to use to run your JAX-RS code. The `web.xml` file provides configuration and deployment information for the web components that comprise a web application. Read about configuring the `web.xml` file for JAX-RS to learn more about this deployment descriptor file.

When using servlets, any servlet path that is defined in the `web.xml` is appended to the base URL. For example, if a root resource has a @javax.ws.rs.Path value of `myresource` and a servlet path of `myservletpath`, the final URL of the resource is `http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>//myservletpath/myresource`.

Procedure

1. Open the `WEB-INF/web.xml` file.
2. Add the following servlet definition to the `WEB-INF/web.xml` file. In the following servlet, you must replace the *unique_servlet_name* with your unique servlet name. Also, replace the *Java_class_name* variable with the full Java package and class name of the javax.ws.rs.core.Application subclass.

```
<servlet>
     <servlet-name>unique_servlet_name</servlet-name>
     <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
     <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>Java_class_name </param-value>
     </init-param>
     <load-on-startup>1</load-on-startup>
</servlet>
```

3. (optional) If there are multiple JAX-RS application subclasses needed in the same web application, you must include an additional servlet initialization parameter, `requestProcessorAttribute`, in the servlet definition that you add to the `WEB-INF/web.xml` file. In the following servlet, you must replace the *unique_servlet_name* with your unique servlet name, the *Java_class_name* variable with the full Java package and class name of the javax.ws.rs.core.Application subclass, and the *unique_identifier* variable with a unique identifier.

```
<servlet>
     <servlet-name>unique_servlet_name_a</servlet-name>
     <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
     <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>Java_class_name_a </param-value>
     </init-param>
     <init-param>
         <param-name>requestProcessorAttribute</param-name>
         <param-value>unique_identifier_a</param-value>
     </init-param>
     <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
     <servlet-name>unique_servlet_name_b</servlet-name>
     <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
     <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>Java_class_name_b</param-value>
     </init-param>
     <init-param>
         <param-name>requestProcessorAttribute</param-name>
         <param-value>unique_identifier_b </param-value>
     </init-param>
     <load-on-startup>1</load-on-startup>
</servlet>
```

4. Add servlet mappings in the `WEB-INF/web.xml` file for each servlet definition. The servlet path is appended to the context root of the web application.

```
<servlet-mapping>
     <servlet-name>servlet_name</servlet-name>
     <url-pattern>servlet_pattern_path</url-pattern>
</servlet-mapping>
```

   For example, if the *servlet_pattern_path* is `/restapi/*`, all valid requests start at the following URL:

```
http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>/restapi/
```

Results

After editing the `WEB-INF/web.xml` file, the web application is configured for the JAX-RS application.

Example

The following example illustrates a `WEB-INF/web.xml` file that configures a servlet path for a JAX-RS application. The servlet path that is defined in the `web.xml` file is appended to the base URL.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4"
xmlns=http://java.sun.com/xml/ns/j2ee
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">


<servlet>
     <servlet-name>RestApplication1</servlet-name>
     <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
     <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>com.ibm.rest.sample.app1.MyApplication</param-value>
     </init-param>
     <init-param>
         <param-name>requestProcessorAttribute</param-name>
```

```
        <param-value>restApplication1ProcessorID</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
    <servlet-name>OtherRestApplicationServlet</servlet-name>
    <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.ibm.rest.other.sample.OtherApplication </param-value>
    </init-param>
    <init-param>
       <param-name>requestProcessorAttribute</param-name>
        <param-value>otherRestApplicationID </param-value>
    </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name> RestApplication1</servlet-name>
    <url-pattern>/rest/api/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>OtherRestApplicationServlet /servlet-name>
    <url-pattern>/other/*</url-pattern>
</servlet-mapping>
</web-app>
```

What to do next

Assemble the web application.

Configuring the web.xml file for JAX-RS filters

The web.xml file contains information about the structure and external dependencies of web components in the module and describes how the components are used at run time. To enable the web container to run Java API for RESTful Web Services (JAX-RS) applications, you can configure the web.xml file to define filters that indicate the possible URLs on which the filter can be invoked.

About this task

You can configure the web.xml file for your web application to enable the JAX-RS application code. You can specify an IBM specific JAX-RS filter to use to run your JAX-RS code. The web.xml file provides configuration and deployment information for the web components that comprise a web application. Read about configuring the web.xml file for JAX-RS to learn more about this deployment descriptor file.

When using servlets, any servlet path defined in the web.xml is appended to the base URL. Filters do not append a path to the resource base URL. Instead, filter URL mappings indicate the possible URLs on which the filter can be invoked. For instance, if a root resource has a @javax.ws.rs.Path value of `myresource`, the final URL of the resource is `http://`*`<your_hostname>`*`:`*`<your Web_container_port>`*`/`*`<context_root_of_Web_application>`*`/myresource`. The URL mapping pattern for the filter must match `myresource` for the root resource to be served correctly. For this example, you can use `/*` or `/myresource` for the URL pattern. When there are multiple resources in the application, the URL pattern for the filter must match all of the resources. The `/*` pattern is a common value for the filter.

If the incoming request URL does not match any JAX-RS resources in the JAX-RS application, the request is passed to the rest of the filter chain. Depending on the application, you might want to use the filter behavior so that requests are served by the JAX-RS application, or if there is no JAX-RS resource available, the request can proceed to an underlying web container artifact, such as a JavaServer Pages (JSP). If the web container has no matching artifact, then the web container is responsible for the error response.

Procedure

1. Open the WEB-INF/web.xml file.

2. Define your filter in the WEB-INF/web.xml file. Add the following filter definition to the WEB-INF/web.xml file. Replace the *unique_filter_name* with your unique filter name. Also replace the *Java_class_name* variable with the full Java package and class name of the JAX-RS application Java subclass.

```
<filter>
        <filter-name>unique_filter_name</filter-name>
        <filter-class>com.ibm.websphere.jaxrs.server.IBMRestFilter</filter-class>
        <init-param>
                <param-name>javax.ws.rs.Application</param-name>
                <param-value>Java_class_name</param-value>
        </init-param>
 </filter>
```

3. (optional) If there are multiple JAX-RS application subclasses needed in the same web application, you must include an additional filter initialization parameter, `requestProcessorAttribute`, in the filter definition that you add to the WEB-INF/web.xml file. In the following filter, replace the *unique_filter_name* with your unique filter name; replace the *Java_class_name* variable with the full Java package and the class name of the JAX-RS application Java subclass; replace the *unique_identifier* variable with a unique identifier.

```
<filter>
   <filter-name>unique_filter_name_a</filter-name>
   <filter-class>com.ibm.websphere.jaxrs.server.IBMRestFilter</filter-class>
   <init-param>
       <param-name>javax.ws.rs.Application</param-name>
       <param-value>Java_class_name_a</param-value>
   </init-param>

   <init-param>
       <param-name>requestProcessorAttribute</param-name>
       <param-value>unique_identifier_a</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>unique_filter_name_b</filter-name>
   <filter-class>com.ibm.websphere.jaxrs.server.IBMRestFilter</filter-class>
   <init-param>
       <param-name>javax.ws.rs.Application</param-name>
       <param-value>Java_class_name_b</param-value>
   </init-param>

   <init-param>
       <param-name>requestProcessorAttribute</param-name>
       <param-value>unique_identifier_b</param-value>
   </init-param>
</filter
```

4. Add filter mappings in the WEB-INF/web.xml file for each filter definition.

   The URL pattern specified in the filter mapping defines to the container the valid URL patterns for invoking the `IBMRestFilter` filter. If an incoming request URL is compatible with the URL pattern, the `IBMRestFilter` is invoked. If the request URL does not match, the filter is not invoked. The request URLs always start at the context root for the filter. See the following example filter mappings:

```
<filter-mapping>
    <filter-name>filter_name</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

   For example, suppose you have the following two resources:

```
@javax.ws.rs.Path("myresource")
public class MyResource {

}
```

```
@javax.ws.rs.Path("myresource2")
public class MyResource2 {

}
```

   You can reach the resources using the following URL:

`http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>/myresource`

`http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>/myresource2`

   If you apply the following filter mapping:

```
<filter-mapping>
        <filter-name>filter_name</filter-name>
        <url-pattern>/myresource</url-pattern>
</filter-mapping>
```

you can use MyResource root resource by visiting the following URL:

```
http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>/myresource
```

This URL invokes the `IBMRestFilter` filter and the URL can find the resource. Because the `/myresource` path matches with the URL pattern in the filter-mapping, the `IBMRestFilter` does get invoked and there is a root resource that has a matching @Path value.

However, suppose you visit the following URL:

```
http://<your_hostname>:<your Web_container_port>/<context_root_of_Web_application>/myresource2
```

the `IBMRestFilter` filter is not invoked because the URL pattern `/myresource` does not match `/myresource2`.

Results

After editing the WEB-INF/web.xml file to apply filters, the web application is configured for the JAX-RS application.

Example

The following example illustrates a WEB-INF/web.xml file that is configured to apply filters to a JAX-RS application. This example defines the `RestApplication1` filter. If an incoming request URL matches a resource in the RestApplication1 filter, the response is generated from RestApplication1 filter. If the incoming request URL does not match a resource in the RestApplication1 filter but matches a resource in OtherRestApplicationFilter, then the response is generated from the OtherRestApplicationFilter filter. If the incoming URL does not match either filter, then the request can be served from another web container artifact, such as a JSP.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4"
xmlns=http://java.sun.com/xml/ns/j2ee
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

     <filter>
        <filter-name>RestApplication1</filter-name>
        <filter-class>com.ibm.websphere.jaxrs.server.IBMRestFilter</filter-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.ibm.rest.sample.app1.MyApplication</param-value>
        </init-param>
        <init-param>
            <param-name>requestProcessorAttribute</param-name>
            <param-value>restApplication1ProcessorID</param-value>
        </init-param>
    </filter>
     <filter>
        <filter-name>OtherRestApplicationServlet</filter-name>
        <filter-class>com.ibm.websphere.jaxrs.server.IBMRestFilter</filter-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.ibm.rest.other.sample.OtherApplication</param-value>
        </init-param>
        <init-param>
            <param-name>requestProcessorAttribute</param-name>
            <param-value>otherRestApplicationID </param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>RestApplication1</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>OtherRestApplicationServlet</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

What to do next

Assemble the web application.

Implementing clients using the Apache Wink REST client

You can use the Apache Wink REST client to send requests and process responses from RESTful services. You can use the client API in Java programs to communicate with web services.

About this task

By default, the Apache Wink client uses the java.net.HttpURLConnection class from the Java runtime environment for issuing requests and processing responses. The Apache Wink client can also use Apache HttpClient 4.0 as the underlying client transport.

You can also use JAX-RS entity providers to help serialize request entities or deserialize response entities. The standard JAX-RS providers used in the JAX-RS server-side services are provided with the client.

You can configure the Apache Wink REST client programmatically or by setting Java Virtual Machine (JVM) properties.

To implement an Apache Wink REST client, you must first create an org.apache.wink.client.ClientConfig object that is then used to construct an org.apache.wink.client.RestClient. You can change the configuration settings for the RestClient object programmatically, or you can use JVM properties to modify the default ClientConfig object values.

To configure the configuration settings for the RestClient object programmatically, invoke the public methods of the ClientConfig object.

Note: After a ClientConfig object is used to construct a RestClient object, the ClientConfig object can no longer be modified. Attempting to do so results in an org.apache.wink.client.ClientConfigException error message.

Alternatively, you can configure the configuration settings for the RestClient object using JVM properties to modify the default ClientConfig object values. Use the following JVM properties to modify the default ClientConfig object values:

wink.client.readTimeout

This property specifies how long the RestClient object waits (in milliseconds) for a response to requests before timing out. A value of zero (0) means that the client waits for an unlimited amount of time and will not timeout.

The default value is 60,000 milliseconds.

wink.client.connectTimeout

This property specifies how long the RestClient object waits (in milliseconds) before timing out when attempting to connect to the target resource. A value of zero (0) means that the client waits for an unlimited amount of time and will not timeout.

The default value is 60,000 milliseconds.

You can programmatically alter any values for the RestClient object that you specify using JVM properties. The programmatic values take precedence over any JVM property values.

Procedure

1. Create an org.apache.wink.client.ClientConfig object.

   The following code snippet illustrates how to create an org.apache.wink.client.ClientConfig object:

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ClientConfig();
```

.

If you use an Apache HTTP client as the underlying transport, create and use an org.apache.wink.client.ApacheHttpClientConfig object instead. You must also include the Apache HTTP client libraries in the classpath. The following code snippet illustrates how to create an org.apache.wink.client.ApacheHttpClientConfig object:

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ApacheHttpClientConfig();
```

2. (optional) Modify the default ClientConfig object values that you want to use for the RestClient object.
   - You can optionally modify the default configuration settings for the RestClient object programmatically. To specify the configuration settings for the RestClient object programmatically, invoke the public methods of the ClientConfig object; for example:

```
clientConfig.connectTimeout(30000);
clientConfig.readTimeout(30000);
```

   - If you are using the Thin Client for JAX-RS in a stand-alone unmanaged client runtime environment, you can optionally modify the configuration settings for the RestClient object using JVM properties. Set the custom JVM properties on the JVM under which the thin client is running.
   - If you are not using the Thin Client for JAX-RS as a stand-alone client runtime environment, but you are using the RestClient object in an application that is intended for installation on the application server, you can optionally modify the configuration settings for the RestClient object using JVM properties. Set the custom JVM properties using the administrative console for your REST client code that is running within an application that is installed on the application server. See the Java virtual machine custom properties information for details on using the administrative console to set these custom JVM properties.
3. (optional) If you use a custom entity provider, add the entity provider using the client configuration.

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ClientConfig();

javax.ws.rs.core.Application app = new javax.ws.rs.core.Application() {
       public Set<Class<?>> getClasses() {
           Set<Class<?>> classes = new HashSet<Class<?>>();
           classes.add(MyCustomEntityProvider.class);
           return classes;
       }
};
clientConfig.applications(app);
```

4. Create a org.apache.wink.client.RestClient object with the client configuration.

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ClientConfig();

javax.ws.rs.core.Application app = new javax.ws.rs.core.Application() {
       public Set<Class<?>> getClasses() {
           Set<Class<?>> classes = new HashSet<Class<?>>();
           classes.add(MyCustomEntityProvider.class);
           return classes;
       }
};
clientConfig.applications(app);

org.apache.wink.client.RestClient client = new org.apache.wink.client.RestClient(clientConfig);
```

5. Create a org.apache.wink.client.Resource object with a URI from the REST client.

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ClientConfig();

javax.ws.rs.core.Application app = new javax.ws.rs.core.Application() {
       public Set<Class<?>> getClasses() {
           Set<Class<?>> classes = new HashSet<Class<?>>();
           classes.add(MyCustomEntityProvider.class);
           return classes;
       }
};
clientConfig.applications(app);

org.apache.wink.client.RestClient client = new org.apache.wink.client.RestClient(clientConfig);

org.apache.wink.client.Resource resource = client.resource("http://www.example.com/rest/api/book/123");
```

6. You can add request headers to the pending request by calling methods on the Resource object.

   You can call a Java method such as post() with the request content as a parameter to send the request. In the following example, an HTTP POST request is made with a Content-Type header value of `text/plain` and an Accept header value of `*/*`.

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ClientConfig();

javax.ws.rs.core.Application app = new javax.ws.rs.core.Application() {
       public Set<Class<?>> getClasses() {
           Set<Class<?>> classes = new HashSet<Class<?>>();
           classes.add(MyCustomEntityProvider.class);
           return classes;
       }
};
clientConfig.applications(app);

org.apache.wink.client.RestClient client = new org.apache.wink.client.RestClient(clientConfig);

org.apache.wink.client.Resource resource = client.resource("http://www.example.com/rest/api/book/123");

ClientResponse response = resource.contentType("text/plain").accept("*/*").post("The request body as a string");
```

Instead of calling `resource.post("The request body as a string")` with a String object, you can use any other object that has a class with a valid javax.ws.rs.ext.MessageBodyWriter object such as a JAXB annotated class, a byte[], or a custom class that has a custom entity provider.

7. Process the response by using the status code, response headers, or the response message body.

```
org.apache.wink.client.ClientConfig clientConfig = new org.apache.wink.client.ClientConfig();

javax.ws.rs.core.Application app = new javax.ws.rs.core.Application() {
       public Set<Class<?>> getClasses() {
           Set<Class<?>> classes = new HashSet<Class<?>>();
           classes.add(MyCustomEntityProvider.class);
           return classes;
       }
};
clientConfig.applications(app);

org.apache.wink.client.RestClient client = new org.apache.wink.client.RestClient(clientConfig);

org.apache.wink.client.Resource resource = client.resource("http://www.example.com/rest/api/book/123");

ClientResponse response = resource.contentType("text/plain").accept("*/*").post("The request body as a string");

System.out.println("The response code is: " + response.getStatusCode());
System.out.println("The response message body is: " + response.getEntity(String.class));
```

Instead of calling the `response.getEntity(String.class)` object with String.class file, you can use any other class that has a valid javax.ws.rs.ext.MessageBodyReader object, such as a JAXB annotated class, a byte[], or a custom class that has a custom entity provider.

8. (optional) Configure the client to transmit basic authentication security tokens. To configure basic authentication for your client, you can choose to manage the appropriate HTTP headers yourself, or more simply, you can use the provided BasicAuthSecurityHandler handler class. The BasicAuthSecurityHandler class simplifies the enablement of basic authentication in the Wink client application. To learn more about using the security client handler to perform basic HTTP authentication, see the securing JAX-RS applications within the web container information.

Results

You have implemented a JAX-RS client using the Apache Wink REST client that can issue requests to a JAX-RS application.

Implementing a client using the unmanaged RESTful web services JAX-RS client

WebSphere Application Server provides a thin Java Platform, Standard Edition 6 (Java SE 6) RESTful web services client runtime to enable application developers to quickly and easily create JAX-RS client applications. The Thin Client for JAX-RS with WebSphere Application Server is a stand-alone Java SE 6 client environment that enables running unmanaged JAX-RS RESTful web services client applications in a non-WebSphere environment to invoke JAX-RS RESTful web services that are hosted by the application server.

Before you begin

Note: You can use the Thin Client for JAX-RS with WebSphere Application Server as a stand-alone client run time in a pure Java SE environment. The Thin Client for JAX-RS running within WebSphere Application Server or WebSphere Application Client environments is not supported. In this version of the application server, other Thin Client run times provided with the application server can also reside in the CLASSPATH and coexist with the Thin Client for JAX-RS.

Before you set up a JAX-RS unmanaged client execution environment, obtain the Thin Client for JAX-RS Java archive (JAR) file. To obtain the Thin Client for JAX-RS, install WebSphere Application Server Version 8.5 or the Application Client for WebSphere Application Server Version 8.5. The Thin Client for JAX-RS JAR file, `com.ibm.jaxrs.thinclient_8.5.0.jar`, is located in the *app_server_root*`\runtimes` directory.

Copy the Thin Client for JAX-RS `com.ibm.jaxrs.thinclient_8.5.0.jar` file to other machines to create a lightweight client environment that enables communications with the product. Copies of the Thin Client for JAX-RS are subject to the same terms and conditions of the license agreement for the WebSphere product where you obtained the Thin Client for JAX-RS. Refer to the license agreements for correct usage and other limitations.

The Thin Client for JAX-RS works with IBM Software Development Kits (SDKs) Version 6.0 and higher. The Thin Client for JAX-RS is also supported on non-IBM software development kits that are V6.0 and higher.

About this task

Set up a Thin Client for JAX-RS environment by completing the following steps.

Procedure

1. Configure the path. Enter the following command to add the Java `bin` directories to your path:

```
set PATH=<your_JDK_bin_directory>;%PATH%
```

2. Configure the class path. Add the Thin Client for JAX-RS JAR file to the classpath definition; for example:

```
set CLASSPATH=.;<your_jax-rs_thin_client_install_directory>\com.ibm.jaxrs.thinclient_8.5.0.jar;
<your_application_jars>;%CLASSPATH%
```

3. Enter the following command to run your client application:

```
%JAVA_HOME%/bin/java <your_client_application>
```

Results

You have set up an unmanaged JAX-RS client runtime environment to invoke RESTful web services that are hosted on a WebSphere Application Server.

Migrating a Feature Pack for Web 2.0 JAX-RS application to WebSphere Version 8

When packaging an application for the Feature Pack for Web 2.0, you must manually make the Java API for RESTful Web Services (JAX-RS) run time available on the application `CLASSPATH`. A common way of doing this is placing the JAX-RS runtime Java archive (JAR) files in the `WEB-INF/lib` directory of the web archive (WAR) module. In Version 8, the JAX-RS runtime environment is native to the application server. It is no longer necessary to manually add the path of the runtime libraries on the `CLASSPATH`. Migrating Web 2.0 applications to the native runtime environment simplifies packaging and deployment, and allows for future updates to take advantage of Java Platform, Enterprise Edition (Java EE) integration features.

Before you begin

Identify Feature Pack for Web 2.0 JAX-RS applications to migrate to the native Version 8 JAX-RS runtime environment. Identify where the JAX-RS runtime JAR files have been made available to the application `CLASSPATH`. Two common methods are the `WEB-INF/lib` directory of the WAR module or by way of a shared library.

About this task

In WebSphere Version 8, the JAX-RS run time runs natively within the application server process. Unlike deploying JAX-RS applications with the Web 2.0 Feature Pack installed on WebSphere Version 7 or earlier, it is no longer necessary to package the JAX-RS runtime JAR files as an external library for the application.

To migrate Web 2.0 applications to the Version 8 runtime environment, simply remove the external JAX-RS runtime libraries. For example, if the runtime JAR files were placed in the `WEB-INF/lib` directory of the WAR module, delete them from that directory before redeploying the application. No other changes are necessary. All other application configurations, such as `web.xml` configuration, are still valid in WebSphere Version 8.

Once the external JAX-RS runtime library is removed, the native runtime environment runs the JAX-RS application. There are advantages to this over Web 2.0 deployment. Application packaging and deployment are simplified since it is no longer necessary to add external libraries on the application `CLASSPATH`. Additionally, new Java EE integration features can be taken advantage of in future updates to the application. Examples are Enterprise JavaBeans (EJB) in a WAR and Java Contexts and Dependency Injection (JCDI) support.

Procedure

1. Remove external JAX-RS runtime libraries from the Feature Pack for Web 2.0 application.
2. Redeploy the application.

Results

You have migrated a Web 2.0 Feature Pack JAX-RS application to the native WebSphere Version 8 JAX-RS runtime environment.

Disabling the JAX-RS runtime environment

There are cases where you must disable the Java API for RESTful Web Services (JAX-RS) runtime environment. When disabling the JAX-RS runtime environment, JAX-RS features are not available, including base JAX-RS runtime capabilities, Enterprise JavaBeans (EJB) runtime integration, Java Contexts and Dependency Injection (JCDI) runtime integration, and Servlet 3.0 web container integration.

About this task

By disabling the JAX-RS runtime environment, any JAX-RS related processing of the application, including processing of classes with scanned JAX-RS annotations, EJB metadata, and JCDI bean enablement, is no longer performed. The JAX-RS runtime environment is not used to process requests and responses to and from the web container.

Note: Disabling the JAX-RS runtime environment does not disable Servlet 3.0 based annotation scanning for JAX-RS annotations such as `javax.ws.rs.Path`. To disable annotation scanning, set the `metadata-complete` attribute. If annotation scanning is disabled, it is disabled for all other components outside of JAX-RS.

The explicit plug points to the IBM JAX-RS runtime environment are the `com.ibm.websphere.jaxrs.server.IBMRestServlet` servlet class and the `com.ibm.websphere.jaxrs.server.IBMRestFilter` filter class. If you specify these classes as your servlet-class or servlet-filter in the web module's `web.xml` file, the IBM JAX-RS runtime environment is used to process requests to that servlet.

To disable the JAX-RS runtime environment from doing so, replace those classes with any other servlet or filter class that can handle expected requests to the servlet, or remove the servlet entirely from the `web.xml` file.

Note: Replacing the `IBMRestServlet` class with another might modify existing behavior in the application. Removing the servlet entirely results in requests not being processed.

Even if not explicitly using the `com.ibm.websphere.jaxrs.server.IBMRestServlet` or `com.ibm.websphere.jaxrs.server.IBMRestFilter` classes, the JAX-RS integration runtime environment may still process the application. For example, if the `web.xml` file of a web module is Servlet 3.0 based, and appropriate conditions are met according to the JSR-311 specification, the JAX-RS integration runtime environment processes scanned classes with JAX-RS annotations and may inject a servlet that can handle requests to the JAX-RS resources in the application.

To disable this functionality, and other functionality such as EJB and JCDI integration, set the `com.ibm.websphere.jaxrs.server.DisableIBMJAXRSEngine` custom Java virtual machine (JVM) property on the application server with a value of `true`.

Procedure

1. Remove references to `IBMRestServlet` and `IBMRestFilter` from the `web.xml` file. The following example illustrates a sample `web.xml` file from an application that uses the IBM JAX-RS runtime environment:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
    version="3.0">
    <servlet>
        <servlet-name>MyRestApplication1</servlet-name>
        <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.ibm.websphere.jaxrs.example.Application1</param-value>
        </init-param>
        <init-param>
            <param-name>requestProcessorAttribute</param-name>
            <param-value>MyRestApplication1RequestProcessorAttribute</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>MyNonJAXRSApplication</servlet-name>
        <servlet-class>com.ibm.websphere.example.NonJAXRSServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <filter>
        <filter-name>MyRestApplication2</filter-name>
        <filter-class>com.ibm.websphere.jaxrs.server.IBMRestFilter</filter-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.ibm.websphere.jaxrs.example.Application2</param-value>
        </init-param>
        <init-param>
            <param-name>requestProcessorAttribute</param-name>
            <param-value>MyRestApplication2RequestProcessorAttribute</param-value>
        </init-param>
    </filter>
    <servlet-mapping>
        <servlet-name>MyRestApplication1</servlet-name>
        <url-pattern>/jaxrsapp1/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>MyNonJAXRSApplication</servlet-name>
```

```
        <url-pattern>/nonjaxrsapp/*</url-pattern>
    </servlet-mapping>
    <filter-mapping>
        <filter-name>MyRestApplication2</servlet-name>
        <url-pattern>/jaxrsapp2/*</url-pattern>
    </filter-mapping>
<web-app>
```

The following example illustrates how the `web.xml` file looks after removing the references to the `IBMRestServlet` and `IBMRestFilter` classes:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
    version="3.0">
    <servlet>
        <servlet-name>MyNonJAXRSApplication</servlet-name>
        <servlet-class>com.ibm.websphere.example.NonJAXRSServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyNonJAXRSApplication</servlet-name>
        <url-pattern>/nonjaxrsapp/*</url-pattern>
    </servlet-mapping>
</web-app>
```

2. Set the `com.ibm.websphere.jaxrs.server.DisableIBMJAXRSEngine` custom JVM property on the application server with a value of `true`.
3. Restart the application server for the custom JVM property to take effect.

Results

You have disabled the JAX-RS runtime environment from processing your application.

Assembling JAX-RS web applications

After you develop the Java class files for your Java API for RESTful Web Services (JAX-RS) web application and edit the web.xml file to enable the JAX-RS servlet, you must assemble the application.

Before you begin

Identify the assembly tool to use to assemble your application. The web application is assembled into a web application archive (WAR) package. You can assemble the WAR package into an enterprise archive (EAR) package if required.

Before assembling the web application, ensure that you have customized the web.xml file to enable the JAX-RS servlet or filter configuration. To learn more, see the information about configuring the web.xml file for the JAX-RS application.

About this task

You must add the JAX-RS libraries to the web application in the WEB-INF/lib directory before you assemble the web application.

By packaging your JAX-RS application classes into the WEB-INF/classes directory of your WAR package and editing the web.xml file, you can use the built-in JAX-RS runtime environment.

Procedure

Create the WAR package using assembly tools.

1. Package your compiled JAX-RS Java classes into the `WEB-INF/classes` directory of your WAR package.
2. Package the web.xml file for your web application in the `WEB-INF/` directory into the WAR package.

Results

A WAR package is created that contains the web application. If needed, you can add the WAR package to an EAR package. The application server can deploy either the WAR or EAR packages.

What to do next

Deploy the web application.

Chapter 34. Developing web services - Security (WS-Security)

The Web Services Security specification defines core facilities for protecting the integrity and confidentiality of a message, and provides mechanisms for associating security-related claims with a message.

Developing applications that use Web Services Security

The Web Services Security specification provides a flexible framework for building secure web services to implement message content integrity and confidentiality. The Web Services Security service programming model supports this flexible framework by providing extension points to integrate new token formats, and methods to obtains keys needed for message protection. The application server programming model provides Web Services Security programming application programming interfaces (WSS API) for securing SOAP messages.

Configuring HTTP basic authentication for JAX-RPC web services programmatically

You can configure HTTP basic authentication for Java API for XML-based RPC (JAX-RPC) web services by programmatically modifying HTTP properties.

Before you begin

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with an assembly tool or with the administrative console.

If you programmatically configure HTTP basic authentication, the properties are configured in the Stub or Call instance. If you choose to configure HTTP basic authentication with the administrative console or an assembly tool, the Web Services Security binding information is modified. The values that are set programmatically take precedence over the values defined in the binding.

About this task

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

Configure HTTP basic authentication programmatically with the following steps.

Procedure

Set the properties in the Stub or Call instance for a Web service or a web service client. You can set properties with the following constant names:

```
javax.xml.rpc.Call.USERNAME_PROPERTY
javax.xml.rpc.Call.PASSWORD_PROPERTY
javax.xml.rpc.Stub.USERNAME_PROPERTY
javax.xml.rpc.Stub.PASSWORD_PROPERTY
```

Example

The following code enables you to configure basic authentication programmatically:

```
Properties prop = new Properties();
InitialContext ctx = new InitialContext(prop);
Service service = (Service)ctx.lookup("java:comp/env/service/StockQuoteService");
QName portQname = new QName("http://httpchannel.test.wsfvt.ws.ibm.com", "StockQuoteHttp");
StockQuote sq = (StockQuote)service.getPort(portQname, StockQuote.class);
((javax.xml.rpc.Stub) sq)._setProperty(javax.xml.rpc.Stub.USERNAME_PROPERTY, "myUser");
((javax.xml.rpc.Stub) sq)._setProperty(javax.xml.rpc.Stub.PASSWORD_PROPERTY, "myPwd");
```

Developing message-level security for JAX-WS web services

JAX-WS applications can be secured with Web Services Security in one of two ways. The application can be secured using policy sets, or through the use of the Web Services Security API (WSS API). The WSS API can only be used to secure a JAX-WS client application. The Web Services Security service programming interface (WSS SPI) provides additional programming interfaces for securing web services.

Web Services Security API programming model

The application server programming model provides Web Services Security programming application programming interfaces (WSS API) for securing SOAP messages.

The API programming model is an interface-based programming model that is based on Web Services Security Version 1.1 standards, but the design also includes support for Web Services Security Version 1.0 for securing SOAP messages. The WSS API programming model implementation is a simplified version, which is based on an early draft proposal of JSR-183, which is the JSR for defining Java API binding for Web Services Security. By design, because the application code is programmed to the interface, any application code that is programmed with the open source implementation should be able to run on the WebSphere Application Server with minimal changes or no changes at all.

The configuration model for web services has also been redesigned from a deployment descriptor model to a policy set model. Web Services Security can be enabled by either using a policy set that is configured by using the administrative console, or by using the WSS API for configuration. The functions provided by the policy set configurations are the same as the functions supported by the WSS API for the Web Services Security run time. However, the security policy that is defined using policy sets has a higher priority over the WSS API. When the WSS API and the policy set are both used in the application, the default behavior is for the security policy from the policy set to be enforced and the WSS API to be ignored. To use the WSS API in the application, you must make sure that there is no policy set attached to the application or to the application resources, or make sure there is no security policy in the attached policy set.

You can still use your existing JAX-RPC applications with Web Services Security; however, those applications cannot take advantage of the Web Services Security Version 1.1 functions, such as configuring the security policy using a policy set, OM filter performance improvements, WSS API, Web Services Secure Conversation (WS-SecureConversation), Kerberos token and the associated SHA-1 key for message protection and identity propagation, and Web Services Trust (WS-Trust) features.

In order to take advantage of the Web Services Security Version 1.1 functions, you must rewrite an existing JAX-RPC application as a JAX-WS application, manually re-configure the security constraints to a policy set, and perform code migration of the DOM-based SPIs to the OM-based SPIs.

For example, when using the JAX-WS programming model, the improved design of the pluggable token framework allows the same security implementation to be used for both the API and policy sets. The framework uses the JAAS Login Module and JAAS Callback Handler for token creation and token validation.

The following diagrams illustrate differences between the programming models.

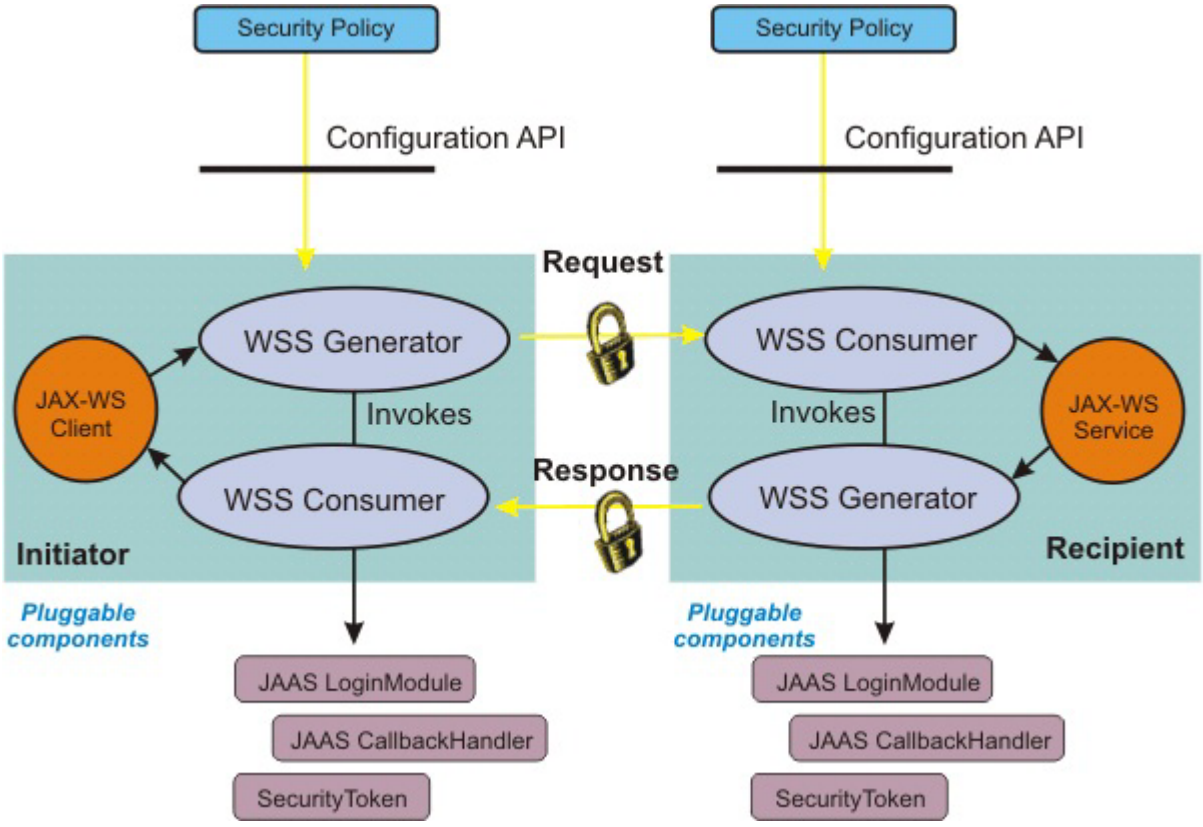


Figure 5. Pluggable token architecture using the JAX-WS programming model

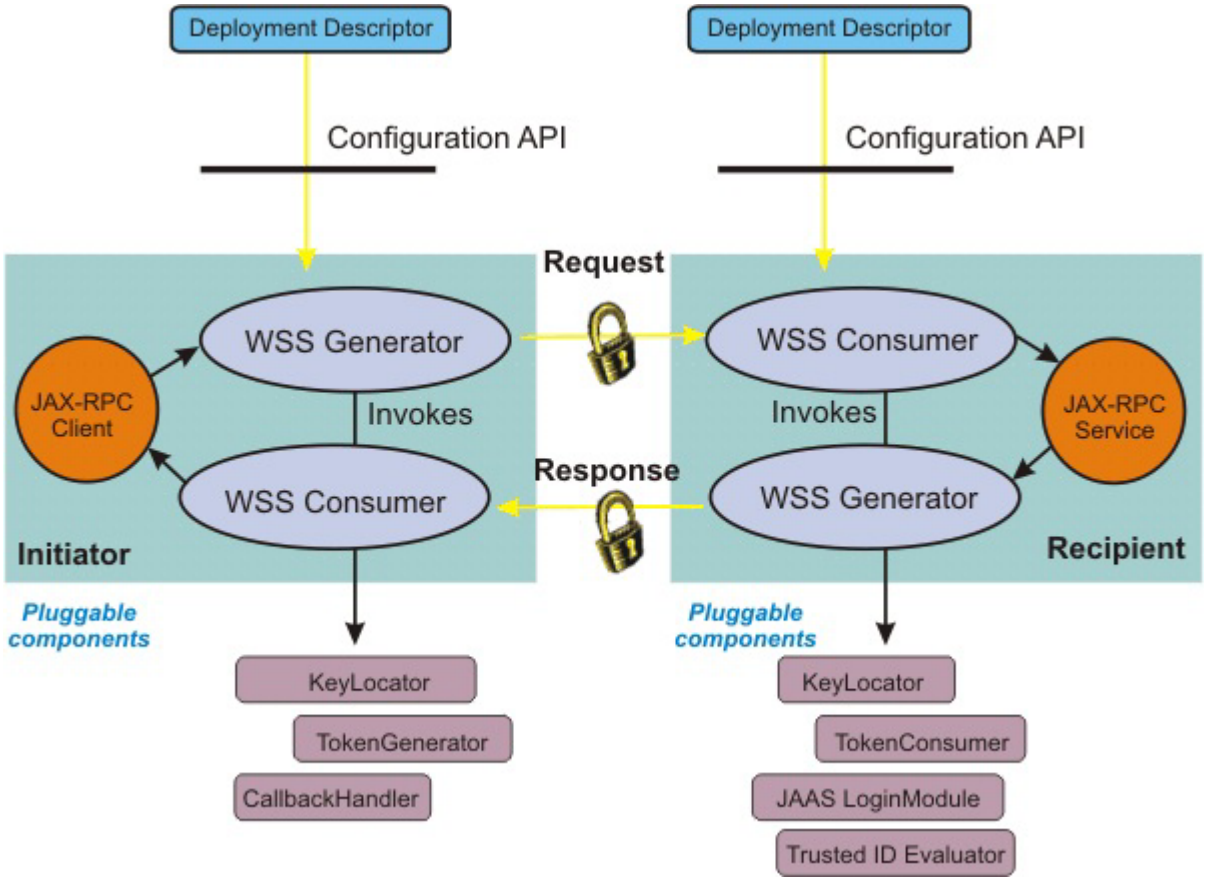


Figure 6. Pluggable token architecture using the JAX-RPC programming model

What is supported when using the WSS APIs

The WSS API can only be used on the client. You can use the Java SE 6 client, the J2EE Application client, or a server client (a service provider acting as client) using the API to secure SOAP message with message-level security.

You should have Web Services Security (WSS) knowledge to use the WSS APIs. Before using the WSS API, keep in mind that the WSS API:

- Are Java-based interfaces.
- Are implemented by using a factory model (WSSFactory).
- Supports the WS-Security Version 1.0 and 1.1 standards, which include the Username and X.509 token profiles, Versions 1.0 and 1.1.
- Are very XML centric.
- Include an object-oriented design which simplifies the APIs.
- Are task oriented and allow common usage scenarios, such as: signing the body and encrypting the SOAP message body content.
- Are flexible and extensible, and they let you to extend the token type support.
- Are based on the provider framework and allow the use of different data models to be used, such as: AXIOM or DOM.
- Provides application programmer with better control and flexibility in applying WSS in their applications.

The default values for the WSS API are predefined and are part of the Web Services Security run time. Default values are provided for:

- The duration of the timestamp
- The signing algorithm, canonicalization algorithm, digest method, transform algorithm, security token reference method and signed parts such as the SOAP body, Web Services Addressing headers and the time stamp.
- The key encryption algorithm, data encryption algorithm, security token reference method, and encrypted parts such as the SOAP body content.

The signature validation has similar default values as the signature (signing information). Similarly, decryption has similar default values as encryption.

What is not supported when using the WSS APIs

The WSS API provided with the application server does not support the following function:

- The application programming model is JAX-WS, meaning JAX-RPC (JSR-109) applications are not supported.
- The WSS API is available in the synchronous message exchange of the JAX-WS client application. However, the WSS API are not supported for the asynchronous client.
- WSS API support is available only for the requester and not for the provider.
- The identity assertion semantic programming model is not supported in the WSS API because identity assertion is not part of the Web Services Security Version 1.0 standard. However, you can use the WSS API to add Identity Assertion semantic in the token processing.

WS-Trust and WS-SecureConversation scenarios

There are several ways to secure the WS-Trust SOAP messages:

- Using the bootstrap policy defined in the policy set.
- Using the WSS API, which supports WS-SecureConversation.
- Enabling dynamic policy for the provider so that the client can retrieve the provider-side policy at run time.

An application would use the WSS API to acquire a security context token for programmatic API-based secure conversation. The WebSphere Application Server trust service provides an application the ability to request a security token for access to a service. The scope and focus of the trust service is only for a WebSphere Application Server Security Context Token (SCT) for WS-SecureConversation.

The WS-SecureConversation and WS-Trust scenarios focus on the inter-operability functions, such as the configuration and runtime interaction of various components. You would use the WSS API to secure the bootstrap RST and RSTR to acquire the security context token from the trust service. After acquiring the security context token, a Derived Key Token is created by using the WSS API. Then the Derived Key Token can be used for signature and encryption.

There are two conditions when using the WSS API to secure the SOAP message with Web Services Security:

- Generation of the secure SOAP message, which is in the request generator application code.
- Consuming of the secured SOAP message, which is in the response consumer application code.

In both cases, a Java exception class com.ibm.websphere.wssecurity.wssapi.WSSException is provided if an error is encountered.

Web services client security context

When the JAX-WS client invokes web services, the current security context that is constructed by the security handler is stored in the RequestContext object. By default, the security context in the JAX-WS

web services client runtime environment is reconstructed for the next web services request invocation. You can preserve the security context for subsequent web services invocations. An example of this is a scenario where the security policy requires the client to send a username security token with the user name and password. When the client sends the first request to invoke the service, you are prompted to enter the required user name and password. The user name and password is saved in a Username SecurityToken token in a Subject in the security context. To avoid being prompted to enter the same user name and password again in subsequent request invocations, you can preserve the security context. There are two methods to preserve the security context: 1) configure the client run time to automatically preserve the client security context for subsequent request invocations; or 2) preserve the security context manually.

To configure the JAX-WS client run time environment to automatically preserve the security context, set the Java system property `com.ibm.websphere.wssecurity.context.management` to `true`. When this system property is true, the JAX-WS client run time copies the security context constructed by the security handler to the RequestContext automatically, and the context is used for subsequent request invocations.

To manually preserve the security context, use the following sample code:

```
// First request
Service svc = Service.create(...);
svc.addPort(...);
Dispatch<String> dispatch = svc.createDispatch(...);
Map<String, Object> requestContext = dispatch.getRequestContext();
String response = dispatch.invoke(body.toString());

Object securityContext = requestContext.get(com.ibm.wsspi.websvcs.Constants.WEBSPHERE_SECURITY_CONTEXT);

// Subsequent request

Dispatch<String> dispatch = svc.createDispatch(...);
Map<String, Object> requestContext = dispatch.getRequestContext();
Object securityContext = requestContext.put(com.ibm.wsspi.websvcs.Constants.WEBSPHERE_SECURITY_CONTEXT, securityContext);
```

Service Programming Interfaces (SPI)

The Web Services Security service programming interface (WSS SPI) provides programming interfaces for securing Web Services Security.

The Web Services Security specification provides a flexible framework for building secure web services to implement message content integrity and confidentiality. The specification does not define specific token formats, but instead associates separate profile documents that define various security token formats and semantics for using those tokens. The Web Services Security service programming model supports the flexible framework by providing extension points to integrate with new token formats, and with methods to obtains keys needed for message protection. Web Services Security uses this programming model to implement support for the standard X.509 token profile, the Username token profile, and the Kerberos token profiles. The programming model is also used to implement support for the LTPA security token, and for new security token types.

The Web Service Security run time token generation and token consuming Service Programming Interfaces (SPI) have been redesigned so that the same security token interface and JAAS Login Module implementation can be used for both the WSS API and the SPI. The WSS SPI for the service provider extends the security token types and provides keys and deriving keys for signing, signature verification, encryption and decryption.

The Web Services Security service programming model provides mechanisms to process custom security tokens, to use custom token in signing and encryption, and to retrieve encryption and signing keys. The Web Services Security service programming interfaces for the JAX-RPC run time, and for the JAX-WS run time, are similar, but not identical.

JAX-RPC run time

The plug-in programming interfaces for the JAX-RPC run time consist of the TokenGenerator, KeyLocator, and JAAS CallbackHandler for outbound message processing, and the TokenConsumer, KeyLocator, and JAAS LoginModule for inbound message processing.

Token Generator, KeyLocator, and Callback Handler
The TokenGenerator class is responsible for formatting the security token to the XML element. This class calls the CallbackHandler class that is specified in the TokenGeneratorConfig object, which obtains the security token input data, and then stores the resulting security token in the Subject object private credentials.

Token Consumer, KeyLocator and JAAS LoginModule
The KeyLocator class is responsible for obtaining the required key for signing and encrypting SOAP message elements from a key store that is specified by the KeyStoreConfig and the KeyLocatorConfig configuration. The TokenConsumer class extracts the token data from the XML security token representation, and stores it in the JAAS Subject using a JAAS LoginModule. The specified KeyLocator class is invoked to find the required key for verifying the digital signature and decrypting the SOAP message elements.

JAX-WS run time

The plug-in programming interfaces for the JAX-WS run time are based on the JAAS programming model for both inbound and outbound SOAP message processing. The JAAS LoginModule and CallbackHandler are responsible for processing the security tokens in SOAP messages. The Login Module and Callback Handler both retrieve and generate tokens, and store the SecurityToken objects in the run time. They replace the functionality of the TokenGenerator, TokenConsumer, and KeyLocator interfaces.

Due to the differences in the programming models, any WebSphere Application Server or custom SPI implementation from the Web Services Security Version 6.1 run time is not supported to run on the Web Services Security run time with the Version 6.1 Feature Pack for Web Services, or the Version 7.0 and later Web Services Security runtime. However, the Web Services Security Version 6.1 run time is supported simultaneously with the Version 6.1 Feature Pack for Web Services, meaning the Version 6.1 SPI implementations are still supported through the original run time. Before using the new Web Services Security run time, a code migration is required to reprogram the Version 6.1 DOM-based SPIs to the AXIOM-based SPIs in the Feature Pack for Web Services, before the SPI can be used.

Developing SAML applications

Use the SAML library application programming interface (API), the SAMLTokenFactory, to configure token parameters, create a SAML token, and bind the created token to a service request. The SAML trust client API provides helper functions that send WS-Trust SOAP requests to the specified external Security Token Service (STS).

About this task

The SAMLTokenFactory API creates SAML tokens through various method signatures. The API also instantiates runtime configuration objects related to the SAML token requester, as well as the recipient.

The WS-Trust Client API for SAML includes the WSSTrustClient class, the WSSTrustClientValidateResult class, and other configuration utility classes.

The following topics provide more information about developing SAML applications using the APIs.

WS-Trust client API:

The WS-Trust client application programming interface (API) includes the WSSTrustClient class, the WSSTrustClientValidateResult class, and other configuration utility classes. The WSSTrustClient class

provides helper functions that send WS-Trust SOAP requests to the specified external Security Token Service (STS) so that the STS can issue or validate one or more SAML assertions and other types of security tokens.

Overview

WebSphere Application Server includes WS-Trust client function, implemented through the WSSTrustClient class, that sends WS-Trust SOAP requests to a specified external Security Token Service (STS). Using the trust requests, the STS can issue one or more SAML assertions or other types of security tokens. The WSSTrustClient class supports the OASIS WS-Trust Version 1.3 specification, and also the WS-Trust Version 1.2 specification. In addition, the SOAP Version 1.1 and SOAP Version 1.2 specifications are supported by the function.

The sample code which follows demonstrates how a web services client uses the WSSTrustClient API to request a SAML bearer token. In the explanatory text which precedes the code sample, the term **SAML token** is used interchangeably with the term **SAML assertion**.

The WSSTrustClient class

You can copy the sample code into an assembly tool application, such as Rational Application Developer, and start using the code after completing the configuration steps. Use the WSSTrustClient class, together with other SAML APIs, to build useful SAML functions. Refer to the SAML API Javadoc for more information.

The WSSTrustClient class is an abstract class and has two concrete implementations: a WS-Trust Version 1.3 implementation and a WS-Trust v1.2 implementation. On line 50 of the code sample, the SAMLWSTrustClientExample web services client code invokes the WSSTrustClient.getInstance(ProviderConfig) method to retrieve the WS-Trust v1.3 implementation. The getInstance() method takes a single ProviderConfig object, which specifies configuration data that are relevant to the SAML token issuer. A ProviderConfig object is also instantiated in the sample code on line 32. The client code sends WS-Trust Version 1.3 request messages to a target STS endpoint. In the sample, the endpoint is https://MyCompany/Trust/13/UsernameMixed. To use the sample code, replace this example STS endpoint with the specific STS endpoint you plan to use.

Note: Starting with WebSphere Application Server Release 8, you can use the `com.ibm.websphere.wssecurity.wssapi.token.SAMLToken` class in Web Services Security (WSS) application programming interface (API). When there is no concern of confusion we use the term `SAMLToken` instead of using its complete package name. You can use WSS API to request SAMLToken processing from an external Security Token Service (STS), to propagate SAMLTokens in SOAP request messages, and to use a symmetric or asymmetric key identified by SAMLTokens to protect SOAP messages.

The WSS API SAML support complements the `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory` and `com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient` interfaces. SAMLTokens that are generated using the `com.ibm.websphere.wssecurity.wssapi.WSSFactory newSecurityToken()` method can be processed by the `SAMLTokenFactory` and `WSSTrustClient` programming interfaces. Conversely, SAMLTokens that are generated by `SAMLTokenFactory` or returned by `WSSTrustClient` can be used in WSS API. Deciding which API to use in your application depends on your specific needs. WSS API SAML support is self contained in the sense that it provides functionality equivalent to that of the `SAMLTokenFactory` and `WSSTrustClient` interfaces as far as web services client applications are concerned. The `SAMLTokenFactory` interface has additional functions to validate SAMLTokens and to create the JAAS Subject that represents authenticated SAMLTokens. This validation is useful for the Web services provider side. When you develop applications to consume SAMLTokens, the `SAMLTokenFactory` programming interface is more suitable for you.

Example: Web services client code that uses the WSSTrustClient class

```
1.  package sample;
2.
3.   import com.ibm.websphere.wssecurity.wssapi.WSSException;
4.   import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
5.   import com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient;
6.   import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
7.   import com.ibm.websphere.wssecurity.wssapi.XMLStructure;
8.
9.
10.  import com.ibm.wsspi.wssecurity.core.token.config.RequesterConfiguration;
11.  import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.Namespace;
12.  import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.TokenType;
13.  import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.WST13;
14.  import com.ibm.wsspi.wssecurity.trust.config.ProviderConfig;
15.  import com.ibm.wsspi.wssecurity.trust.config.RequesterConfig;
16.  import com.ibm.wsspi.wssecurity.wssapi.OMStructure;
17.
18.  import org.apache.axiom.om.OMElement;
19.  import org.apache.axis2.util.XMLPrettyPrinter;
20.
21.  import java.util.List;
22.  import java.io.ByteArrayOutputStream;
23.  import java.io.InputStream;
24.  import java.io.BufferedReader;
25.  import java.io.InputStreamReader;
26.  import java.io.IOException;
27.
28.  public class WSSTrustClientExample {
29.
30.    public static void main(String[] args) {
31.          try {
32.          ProviderConfig providerConfig = WSSTrustClient.newProviderConfig(Namespace.WST13, https://MyCompany.com/Trust/13/UsernameMixed );
33.
34.          showProviderConfigDefaultValue(providerConfig);
35.
36.          providerConfig.setPolicySetName("Username WSHTTPS default");
37.          providerConfig.setBindingName("SamlTCSample");
38.          providerConfig.setBindingScope("domain");
39.
40.
41.          RequesterConfig requesterConfig = WSSTrustClient.newRequesterConfig(Namespace.WST13);
42.
43.          showRequestConfigDefaultValue(requesterConfig);
44.
45.          requesterConfig.put(RequesterConfiguration.RSTT.APPLIESTO_ADDRESS, "https://user.MyCompany:9443/WSSampleSei/EchoService12");
46.          requesterConfig.put(RequesterConfiguration.RSTT.TOKENTYPE, TokenType.SAML11);
47.          requesterConfig.put(RequesterConfiguration.RSTT.KEYTYPE, WST13.KEYTYPE_BEARER);
48.          requesterConfig.setSOAPNamespace(Namespace.SOAP12);
49.
50.          WSSTrustClient client = WSSTrustClient.getInstance(providerConfig);
51.          List<SecurityToken> securityTokens = client.issue(providerConfig, requesterConfig);
52.
53.          // Process SAML token
54.          if (securityTokens != null && !securityTokens.isEmpty()) {
55.            System.out.println("Number of tokens returned = " + securityTokens.size());
56.            SecurityToken token = securityTokens.get(0);
57.            if (token instanceof SAMLToken) {
58.              showSAMLToken((SAMLToken)token);
59.            } else {
60.              System.out.println("Returned token is not an SAMLToken");
61.            }
62.          } else {
63.            System.out.println("No securityToken obtained.");
64.          }
65.
66.       } catch (SoapSecurityException ex) {
67.         System.out.println("Caught exception: " + ex.getMessage());
68.         ex.printStackTrace();
69.       }
70.    }
71.
72.    private static void showProviderConfigDefaultValue(ProviderConfig providerConfig) {
73.      System.out.println("providerConfig.getApplicationName() = " + providerConfig.getApplicationName());
74.      System.out.println("providerConfig.getBindingName() = " + providerConfig.getBindingName());
75.      System.out.println("ProviderConfig.getBindingScope() = " + providerConfig.getBindingScope());
76.      System.out.println("providerConfig.getIssuerURI() = " + providerConfig.getIssuerURI());
77.
78.      System.out.println("providerConfig.getPolicySetName() = " + providerConfig.getPolicySetName());
79.      System.out.println("ProviderConfig.getPortName() = " + providerConfig.getPortName());
80.      System.out.println("providerConfig.getProvider() = " + providerConfig.getProvider());
81.      System.out.println("ProviderConfig.getServiceName() = " + providerConfig.getServiceName());
82.      System.out.println("providerConfig.getWSTrustNamespace() = " + providerConfig.getWSTrustNamespace());
83.      System.out.println("ProviderConfig.toString() = " + providerConfig.toString());
84.    }
85.
86.    private static void showRequestConfigDefaultValue(RequesterConfig requesterConfig) {
87.      System.out.println("requesterConfig.getRSTTProperties() = " + requesterConfig.getRSTTProperties());
88.      System.out.println("requesterConfig.getSecondaryParameters() = " + requesterConfig.getSecondaryParameters());
```

```
89.      System.out.println("requesterConfig.getSOAPNamespace() = " + requesterConfig.getSOAPNamespace());
90.      System.out.println("requesterConfig.getWSAddressingNamespace() = " + requesterConfig.getWSAddressingNamespace());
91.
92.      System.out.println("requesterConfig.getMessageID() = " + requesterConfig.getMessageID());
93.      System.out.println("requesterConfig.toString() = " + requesterConfig.toString());
94.    }
95.
96.    private static void showSAMLToken(SAMLToken samlToken){
97.      System.out.println("samlToken.getAssertionQName() = " + samlToken.getAssertionQName());
98.      System.out.println("samlToken.getAudienceRestriction() = " + samlToken.getAudienceRestriction());
99.      System.out.println("samlToken.getAuthenticationMethod() = " + samlToken.getAuthenticationMethod());
100.     System.out.println("samlToken.getConfirmationMethod() = " + samlToken.getConfirmationMethod());
101.     System.out.println("samlToken.getId() = " + samlToken.getId());
102.     System.out.println("samlToken.getKeyIdentifier() = " + samlToken.getKeyIdentifier());
103.     System.out.println("samlToken.getKeyIdentifierEncodingType() = " + samlToken.getKeyIdentifierEncodingType());
104.     System.out.println("samlToken.getKeyIdentifierValueType() = " + samlToken.getKeyIdentifierValueType());
105.     System.out.println("samlToken.getKeyName() = " + samlToken.getKeyName());
106.     System.out.println("samlToken.getPrincipal() = " + samlToken.getPrincipal());
107.     System.out.println("samlToken.getProperties() = " + samlToken.getProperties());
108.     System.out.println("samlToken.getReferenceURI() = " + samlToken.getReferenceURI());
109.     System.out.println("samlToken.getSAMLAttributes() = " + samlToken.getSAMLAttributes());
110.     System.out.println("samlToken.getSamlCreated() = " + samlToken.getSamlCreated());
111.     System.out.println("samlToken.getSamlExpires() = " + samlToken.getSamlExpires());
112.     System.out.println("samlToken.getSamlID() = " + samlToken.getSamlID());
113.     System.out.println("samlToken.getSAMLIssuerName() = " + samlToken.getSAMLIssuerName());
114.     System.out.println("samlToken.getSAMLNameID() = " + samlToken.getSAMLNameID());
115.     System.out.println("samlToken.getStringAttributes() = " + samlToken.getStringAttributes());
116.     System.out.println("samlToken.getSubjectDNS() = " + samlToken.getSubjectDNS());
117.     System.out.println("samlToken.getSubjectIPAddress() = " + samlToken.getSubjectIPAddress());
118.     System.out.println("samlToken.getThumbprint() = " + samlToken.getThumbprint());
119.     System.out.println("samlToken.getThumbprintEncodingType() = " + samlToken.getThumbprintEncodingType());
120.     System.out.println("samlToken.getThumbprintValueType() = " + samlToken.getThumbprintValueType());
121.     System.out.println("samlToken.getTokenQname() = " + samlToken.getTokenQname());
122.     System.out.println("samlToken.getValueType() = " + samlToken.getValueType());
123.
124.    XMLStructure samlXmlStructure = samlToken.getXML();
125.     if (samlXmlStructure != null && samlXmlStructure instanceof OMStructure) {
126.     OMStructure samlOMStructure = (OMStructure) samlXmlStructure;
127.     System.out.println("((OMStructure)samlToken.getXML()).getNode()formatted = " + formatXML(samlOMStructure.getNode()));
128.    }
129.
130.    try {
131.      InputStream is = samlToken.getXMLInputStream();
132.      if (is != null) {
133.        try {
134.          BufferedReader reader = new BufferedReader(new InputStreamReader(is));
135.          StringBuilder sb = new StringBuilder();
136.          String line = null;
137.          while ((line = reader.readLine()) != null) {
138.            sb.append(line + "\n");
139.          }
140.          System.out.println(sb.toString());
141.        } catch (Exception ex) {
142.          System.out.println("Caught exception reading from InputStream: " + ex.getMessage());
143.          ex.printStackTrace();
144.        } finally {
145.          try {
146.            is.close();
147.          } catch (IOException e) {
148.            e.printStackTrace();
149.          }
150.        }
151.      }
152.    } catch (WSSException wex) {
153.      System.out.println("Caught exception getXMLInputStream(): " + wex.getMessage());
154.      wex.printStackTrace();
155.    }
156.  }
157.
158.  private static String formatXML(OMElement omInput) {
159.    ByteArrayOutputStream out = new ByteArrayOutputStream();
160.    String output = "";
161.
162.    try {
163.          XMLPrettyPrinter.prettify(omInput, out);
164.          output = out.toString();
165.    } catch (Throwable e) {
166.        try {
167.            output = omInput.toString();
168.        } catch (Throwable e2) {
169.                System.out.println("Caught exception: " + e2.getMessage());
170.        e2.printStackTrace();
171.        }
172.        }
173.    return output;
174.  }
175.
176.  }
```

WSSTrustClient class support for policy sets and bindings

The WS-Trust client function supports both application-specific bindings and general bindings for use with the trust client policy set and binding documents. In addition, general bindings and default bindings are supported if the application is running in the application server environment. General bindings are supported in the thin client environment, but default bindings are not.

Managing the policy set and bindings for the WS-Trust client API is similar to managing a policy set and bindings for a web services client. However, differences exist that are unique to the WS-Trust client. One difference is that the WS-Trust client does not use policy set attachments. Instead, the policy set name and binding name are specified in a ProviderConfig object, as shown in line 36 and line 37 of the sample code.

When the WS-Trust client looks for bindings, the way the client manages the search scope differs from the web services client. If you do not specify the wstrustClientBindingScope property for the trust client binding, the system first searches the application for an application-specific binding with the binding name that you specified. If a binding is found, it is used for the trust client request. If no application-specific binding is found, the system searches the available general bindings for a binding with the name that you specified. If a general binding is found, it is used for the trust client request. If no bindings with the specific name are found, then default bindings are used in a server environment. Default bindings are only used in a server environment. If the binding scope is specified, only that scope is used for the binding search.

Line 38 of the sample code, `providerConfig.setBindingScope("domain")`, indicates that the example uses general bindings. You can also set the binding scope to `application` to indicate that the sample code uses application-specific bindings. The example uses the general binding named **SamlTCSample**. Both application-specific and general bindings are supported in the application server and the thin client environment. For more information about configuring the SamlTCSample bindings when the application is installed on the application server, read about configuring policy sets and bindings to communicate with STS.

The showProviderConfigDefaultValue(providerConfig) code on line 34 of the sample code shows the default settings. The sample code includes a utility method that prints out the contents of providerConfig.

Line 51 of the sample code, List<SecurityToken> securityTokens = client.issue(providerConfig, requesterConfig), sends an issue WS-Trust request. The second parameter on this line specifies the RequesterConfig object, and this parameter determines the content of the issue request. The code on line 41, RequesterConfig requesterConfig = WSSTrustClient.newRequesterConfig(Namespace.WST13), instantiates a RequesterConfig object that is used to construct the trust request using the WS-Trust Version 1.3 namespace. A utility function is shown on line 43: showRequestConfigDefaultValue(requesterConfig). This function displays the default settings for the RequesterConfig object. The code between lines 45 and 48 initializes the RequesterConfig to request a Version 1.1 SAML bearer token. This token is used to access the service endpoint using the SOAP 1.2 namespace. In the example, the service endpoint is https://user.MyCompany.com:9443/WSSampleSei/EchoService12.

JVM arguments support

Before executing the sample code, you must set up several Java Virtual Machine (JVM) arguments. The sample code implements the Username WSHTTPS default policy set, which has two requirements: 1) a Username token is sent to the STS; and 2) messages are protected using Secure Sockets Layer (SSL). To set up the environment to meet these requirements, first configure the `ssl.client.props` file to define a truststore. For step-by-step instructions, read about running an unmanaged web services JAX-WS client.

To meet the second requirement regarding SSL message protection, obtain a copy of the STS SSL X.509 certificate and inset it into the truststore. To do this, follow the steps in the topic, Using the retrieveSigners command in SSL, to enable server-to-server trust. Alternately, you can accept the STS certificate when

you send the first trust request to the STS if the com.ibm.ssl.enableSignerExchangePrompt property in the profile_home/properties/ssl.client.props file is set to true. For more information about this option, read about changing the signer auto-exchange prompt at the client.

In addition, you must specify the client JAAS configuration file so that the client runtime environment can locate the Username token LoginModule JAAS login configuration. Specify the parameter using this code: -Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf. You must also include the thin client jar, for example com.ibm.jaxws.thinclient_8.0.0.jar, in the classpath. For more information, read about running an unmanaged web services JAX-WS client application.

Sample code execution

A prerequisite to executing the sample code is to set up an external STS endpoint to issue a SAML 1.1 bearer token for the specified web services as defined by the RequesterConfiguration.RSTT.APPLIESTO_ADDRESS property.

Executing the sample code generates a WS-Trust issue request message, as shown in the example:

```
177.  <?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
178.      <soapenv:Header>
179.          <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing">https://user.MyCompany.com/Trust/13/UsernameMixed</wsa:To>
180.          <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing">urn:uuid:4951B6775950CAC92A1252458259166</wsa:MessageID>
181.          <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing">http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue</wsa:Action>
182.      </soapenv:Header>
183.      <soapenv:Body>
184.          <wst:RequestSecurityToken xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
185.              <wst:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1</wst:TokenType>
186.              <wst:RequestType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue</wst:RequestType>
187.              <wst:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</wst:KeyType>
188.              <wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
189.                  <wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/addressing">
190.                      <wsa:Address>https://user.MyCompany.com:9443/WSSampleSei/EchoService12</wsa:Address>
191.                  </wsa:EndpointReference>
192.              </wsp:AppliesTo>
193.          </wst:RequestSecurityToken>
194.      </soapenv:Body>
195.  </soapenv:Envelope>
```

To view the WS-Trust request message, you must enable a client-side trace. Set the following JVM properties:

- -DtraceSettingsFile=MyTraceSettings.properties
- -Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager
- -Djava.util.logging.configureByServer=true

For more information about these properties, read about enabling trace on client and stand-alone applications. In addition to setting the JVM properties, you must also specify the trace setting, com.ibm.ws.wssecurity.*=all=enabled, in the MyTraceSettings.properties file. Look for Trust Client outgoing request: in the trace log file.

SAML token return

The code on line 51 of the sample code, List<SecurityToken> securityTokens = client.issue(providerConfig, requesterConfig), returns a SAML token if the WS-Trust issue request is processed successfully. The code between lines 54 and 64 processes the returned SAML token. The utility function shown on line 58, showSAMLToken((SAMLToken)token), displays the content of the received SAML token. The showSAMLToken() routine shows the SAML token as an XML document. An example of this XML document is provided in line 196 to line 233 of the sample code.

```
196.  <?xml version="1.0" encoding="UTF-8"?>
197.  <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion" MajorVersion="1" MinorVersion="1"
198.   AssertionID="_f7f65d28-fbb1-4e10-8ddf-f4b6ed0c8277" Issuer="http://MyCompany.com/Trust"
199.   IssueInstant="2009-09-09T01:04:41.144Z">
200.      <saml:Conditions NotBefore="2009-09-09T01:04:41.141Z" NotOnOrAfter="2009-09-09T11:04:41.141Z">
201.          <saml:AudienceRestrictionCondition>
202.              <saml:Audience>https://user.MyCompany.com:9443/WSSampleSei/EchoService12</saml:Audience>
203.          </saml:AudienceRestrictionCondition>
204.      </saml:Conditions>
205.      <saml:AuthenticationStatement AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
206.       AuthenticationInstant="2009-09-09T01:04:41.131Z">
207.          <saml:Subject>
208.              <saml:SubjectConfirmation>
```

```
209.                <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
210.            </saml:SubjectConfirmation>
211.        </saml:Subject>
212.    </saml:AuthenticationStatement>
213.    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
214.        <ds:SignedInfo>
215.            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
216.            <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
217.            <ds:Reference URI="#_f7f65d28-fbb1-4e10-8ddf-f4b6ed0c8277">
218.                <ds:Transforms>
219.                    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
220.                    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
221.                </ds:Transforms>
222.                <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
223.                <ds:DigestValue>AQ6e7YQqKgcg/B/ebBj8/DF+uWg=</ds:DigestValue>
224.            </ds:Reference>
225.        </ds:SignedInfo>
226.        <ds:SignatureValue>SuccIOniR . . . . yjTh9iQs=</ds:SignatureValue>
227.        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
228.            <X509Data>
229.                <X509Certificate>MIIB3zCCAUi . . . . itzymqg3</X509Certificate>
230.            </X509Data>
231.        </KeyInfo>
232.    </ds:Signature>
233. </saml:Assertion>
```

SAML token library APIs:

The SAML token library application programming interfaces (APIs) provide methods you can use to create, validate, parse, and extract SAML tokens.

Overview

The library implementation for SAML Version 1.1 and SAML Version 2.0 provides three types of subject confirmation: holder-of-key (HoK), bearer, and sender-vouches. You can use the SAML token library APIs to create, validate, and extract the attributes of a SAML HoK or bearer token. SAML token propagation from web services SOAP messages is also discussed. Sample code is provided to demonstrate the use of the APIs.

WebSphere Application Server with SAML provides default policy sets to support the bearer and HoK subject confirmation.

These sections discuss creating a SAML token using the SAML token library APIs:

1. “Configuration of token creation parameters” on page 1476
2. “SAML token factory instance creation” on page 1477
3. “SAML token creation” on page 1477
4. “Sample code” on page 1478

The SAMLTokenFactory API is the primary SAML token library programming interface. SAMLTokenFactory supports creating, parsing, and validating both SAML 1.1 and SAML 2.0 tokens. Using the SAMLTokenFactory API, you can create ProviderConfid, RequesterConfig, and ConsumerConfig configuration objects to define the required SAML token characteristics. Read the API documentation for more details.

You can perform additional operations on a SAML token after it is created, including:

- “SAML token validation” on page 1477
- “SAML token identity mapped to a subject” on page 1477
- “Parse assertion elements” on page 1478
- “SAML token attributes extraction” on page 1478

Note: Starting with WebSphere Application Server Release 8, you can use the `com.ibm.websphere.wssecurity.wssapi.token.SAMLToken` class in Web Services Security (WSS) application programming interface (API). When there is no concern of confusion we use the term `SAMLToken` instead of using its complete package name. You can use WSS API to request

SAMLToken processing from an external Security Token Service (STS), to propagate SAMLTokens in SOAP request messages, and to use a symmetric or asymmetric key identified by SAMLTokens to protect SOAP messages.

The WSS API SAML support complements the `com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory` and `com.ibm.websphere.wssecurity.wssapi.trust.WSSTrustClient` interfaces. SAMLTokens that are generated using the `com.ibm.websphere.wssecurity.wssapi.WSSFactory newSecurityToken()` method can be processed by the `SAMLTokenFactory` and `WSSTrustClient` programming interfaces. Conversely, SAMLTokens that are generated by `SAMLTokenFactory` or returned by `WSSTrustClient` can be used in WSS API. Deciding which API to use in your application depends on your specific needs. WSS API SAML support is self contained in the sense that it provides functionality equivalent to that of the `SAMLTokenFactory` and `WSSTrustClient` interfaces as far as web services client applications are concerned. The `SAMLTokenFactory` interface has additional functions to validate SAMLTokens and to create the JAAS Subject that represents authenticated SAMLTokens. This validation is useful for the Web services provider side. When you develop applications to consume SAMLTokens, the `SAMLTokenFactory` programming interface is more suitable for you.

Configuration of token creation parameters

When you configure the token creation parameters, the configuration information relates to either the requesting entity, the issuing entity, or the receiving entity. In this example, configuration information is defined for the requesting and the issuing entities. For each type of supported subject confirmation, the SAML token library provides pre-configured attributes for the requesting entity. These attributes are used during the creation of the self-issued SAML token by the WebSphere runtime environment. A self-issued SAML token is one that is generated locally, instead of one that is requested from a Security Token Service (STS). If you need to customize the attributes for a default parameter, use the RequesterConfig parameter. For more information, read about the RequesterConfig parameter in the SAMLTokenFactory API topic.

First, set up the requestor configuration information:

```
// Setup the requester's configuration information (parameters needed
// to create the token specified as configuration properties).
// in this case we are using the configuration information to create a
// SAML token that contains a symmetric holder of key subject
// confirmation.
RequesterConfig requesterData =
     samlFactory.newSymmetricHolderOfKeyTokenGenerateConfig();
```

Next, set the recipient public key alias and optionally, the authentication method:

```
// Set recipient's public key alias
// (in this example we use SOAPRecipient), so the provider can encrypt secret
// key for the receiving end.
requesterData.setKeyAliasForAppliesTo("SOAPRecipient");

// Set the authentication method that took place. This is an optional
// parameter.
reqData.setAuthenticationMethod("Password");
```

Then set the issuer configuration attributes:

```
// Set issuer information by instantiating a default ProviderConfig.
// See javadocs for the SAMLTokenFactory class on the details of the
// default values and how to modify them.
ProviderConfig samlIssuerCfg =
  samlFactory.newDefaultProviderConfig("WebSphereSelfIssuer");
```

SAML token factory instance creation

Use the SAMLTokenFactory class, specifying the SAML token type, either Version 1.1 or Version 2.0. Set additional parameters for creating the SAML token.

Use the SAMLTokenFactory class with the SAML token type:

```
// Instantiate a token factory based on the version level of the token
// to use. In this example we use the SAML v1.1 token factory.
SAMLTokenFactory samlFactory =
SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

Set additional parameters in the CredentialConfig object using the caller subject or the runAsSubject:

```
// Retrieve the caller subject or the runAsSubject (depending on your
// scenario) then use the Subject to get a CredentialConfig object
// using the SAML token library.
// This invocation requires the
// wssapi.SAMLTokenFactory.newCredentialConfig" Java Security
// permission.
CredentialConfig cred = samlFactory.newCredentialConfig(runAsSubject);
```

SAML token creation

Create the SAML token using the token factory:

```
// Now create the SAML token. This invocation requires the
// "wssapi.SAMLTokenFactory.newSAMLToken" Java Security permission.
SecurityToken samlToken =
                samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

SAML token validation

An entity that receives a SAML token, such as a business service, can use the SAML token library API to validate the token before using it. For example, the service needs to validate the token before extracting the SAML attributes from the requester. An existing SAML assertion document can be validated using the configuration data from the consumer.

The following API code validates the token:

```
ConsumerConfig consumerConfig = samlFactory.newConsumerConfig();
 XMLStructure xml =
 try {
    SAMLToken token = samlFactory.newSAMLToken( consumerConfig, XMLStructure xml );
    // token successfully validated
 } catch(WSSException e){
  // token failed validation
 }
```

SAML token identity mapped to a subject

A SAML token can be used to create a subject. The name identifier in the SAML token is mapped to a user in the user registry to obtain the principal name for the subject.

```
Subject subject;
SAMLToken aSAMLToken = ...;
  try {
        subject = samlFactory.newSubject(aSAMLToken);
  } catch(WSSException e) {
  }
```

Parse assertion elements

The recipient of a SAML token can parse and extract assertion elements from the SAML token using the SAMLToken APIs, which are included in the SAML token library API. For example, the token creation time can be extracted using this code:

```
Date dateCreated = samlToken.getSamlCreated();
```

Extract the name of the token issuer and the confirmation method as follows:

```
String confirmationMethpo = samlToken.getConfirmationMethod();
String issuerName = samlToken.getSAMLIssuerName();
```

If the extracted subject confirmation method is returned as holder-of-key confirmation, then you can use the following API to retrieve the bytes for the key material:

```
byte[] hokBytes = samlToken.getHolderOfKeyBytes();
```

For more information about all the SAML APIs, read the API documentation for the SAMLToken interface.

SAML token attributes extraction

Extract SAML attributes from the initiating entity (service requester) using the SAMLToken API, as shown in the following code snippets.

```
// Get all attributes
List<SAMLAttribute> allAttributes =
             ((SAMLToken) samlToken).getSAMLAttributes();

// Iterate over the attribute and process accordingly
Iterator<SAMLAttribute> iter = allAttributes.iterator();
while (iter.hasNext())
{
    SAMLAttribute anAttribute = iter.next();

    // Handle attributes
    String attributeName = anAttribute.getName();
    String[] attributeValues = anAttribute.getStringAttributeValue();

}
```

Sample code

The sample code demonstrates how to use the SAML token library APIs to accomplish some of the operations previously described. A JVM property that points to the location of the SAML properties file is a prerequisite for running this code. The SAML properties file, `SAMLIssuerConfig.properties`, must contain configuration attributes related to the issuer (provider) of the SAML token.

The default location of the SAMLIssuerConfig.properties file for the cell level is: *app_server_root*/profiles/*$PROFILE*/config/cells/*$CELLNAME*/sts.

For the server level, the default location is: *app_server_root*/profiles/*$PROFILE*/config/cells/*$CELLNAME*/nodes/*$NODENAME*/servers/*$SERVERNAME*.

```
IssuerURI=WebSphere
TimeToLive=3600000
KeyStorePath=c:/samlsample/saml-provider.jceks
KeyStoreType=jceks
KeyStorePassword=myissuerstorepass
KeyAlias=samlissuer
KeyName=CN=SAMLIssuer, O=IBM, C=US
KeyPassword=xxxxxxxxx
TrustStorePath=c:/samlsample/saml-provider.jceks
TrustStoreType=jceks
TrustStorePassword=yyyyyyyy

package samlsample;
```

```
import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;
import javax.security.auth.Subject;

// Import methods from the SAML token library
import com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.wsspi.wssecurity.saml.config.ProviderConfig;
import com.ibm.wsspi.wssecurity.saml.config.RequesterConfig;
import com.ibm.wsspi.wssecurity.saml.config.CredentialConfig;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.RequesterConfiguration;

public class SamlAPIsample {

public void testSAMLTokenLibrary() throws Exception {

try {
  // Get an instance of the SAML v1.1 token factory
  SAMLTokenFactory samlFactory = SAMLTokenFac
  tory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

  // Generate default requester data for a subject confirmation of
  // type holder-of-key (secret key).
  RequesterConfig requesterData =
       samlFactory.newSymmetricHolderOfKeyTokenGenerateConfig();

  // Set the recipient's key alias, so that the issuer can encrypt
  // the secret key for recipient as part of the subject confirmation.
  requesterData.setKeyAliasForAppliesTo("SOAPRecipient");

  // Set the authentication method that took place.
  requesterData.setAuthenticationMethod("Password");

  System.out.println("default holder of key confirmation key type is: "+
    Requester
  Data.getRSTTProperties().get(RequesterConfiguration.RSTT.KEYTYPE));
  Requester
  Data.getRSTTProperties().put(RequesterConfiguration.RSTT.KEYTYPE,
      "http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey");

  requesterData.getRSTTProperties().put(
                      RequesterConfiguration.RSTT.APPLIESTO_ADDRESS,

                      "http://localhost:9080");

  requesterData.setConfirmationMethod("holder-of-key");

  // Set the recipient's key alias so that token infomation such as
  // the secret HoK can be encrypted by the issuer and decrypted by the
  // recipient.
  requesterData.setKeyAliasForAppliesTo("SOAPRecipient");
  requesterData.setAuthenticationMethod("Password");
  requesterData.getRSTTProperties().put(
  RequesterConfiguration.RSTT.ENCRYPTIONALGORITHM,
                       "http://www.w3.org/2001/04/xmlenc#aes128-cbc");
  requester
  Data.getRSTTProperties().put(RequesterConfiguration.RSTT.TOKENTYPE,
      "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
       1.1#SAMLV1.1");
  requesterData.setRequesterIPAddress("9.53.52.65");

  // Print requester configuration items
  System.out.println("authentication method for requester is: "+
             requesterData.getAuthenticationMethod());
  System.out.println("confirmation method for requester is: "+
                    requesterData.getConfirmationMethod());
  System.out.println("key alias for requester is: "+
            requesterData.getKeyAliasForRequester());
  System.out.println("key alias for recipient as set in requester config is "+
          requesterData.getKeyAliasForAppliesTo());
  System.out.println("holder of key confirmation key type is: "+
             Requester
  Data.getRSTTProperties().get(RequesterConfiguration.RSTT.KEYTYPE));

  // Get an instance of the Credential config object
  CredentialConfig cred = samlFactory.newCredentialConfig();
  cred.setRequesterNameID("Alice");

  // Set some user attributes
  ArrayList<SAMLAttribute> userAttrs = new ArrayList<SAMLAttribute>();
  SAMLAttribute anAttribute = new SAMLAttribute("EmployeeInfo",
      new String[] {"GreenRoofing","JohnDoe", "19XY981245"},
                null, "WebSphere Namespace", null, "JohnDoeInfo " );
  userAttrs.add(anAttribute);
  cred.setSAMLAttributes(userAttrs);

  // Get default provider configuration
```

```
  ProviderConfig samlIssuerCfg =
    samlFactory.newDefaultProviderConfig("WebSphereSelfIssuer");
  System.out.println("time to live from the default provider config: "+
            samlIssuerCfg.getTimeToLive());
  System.out.println("keyStore path from default provider config: "+
                    samlIssuerCfg.getKeyStoreConfig().getPath());
  System.out.println("keyStore type from default provider config: "+
                   samlIssuerCfg.getKeyStoreConfig().getType());
  System.out.println("key alias from default provider config: "+
                   samlIssuerCfg.getKeyInformationConfig().getAlias());

  // Generate the SAML token
  SecurityToken samlToken =
         samlFactory.newSAMLToken(cred, requesterData, samlIssuerCfg);
  System.out.println("token's creation Date is:
                "+((SAMLToken)samlToken).getSamlCreated().toString());
  System.out.println("token's expiration Date is:
               "+((SAMLToken)samlToken).getSamlExpires().toString());
  System.out.println("token's subject confirmation method is:
                 "+((SAMLToken)samlToken).getConfirmationMethod());


  // Create a Subject, mapping the name identifier in the token to a user
  // in the user registry to obtain the Principal name
  Subject subject = samlFactory.newSubject((SAMLToken)samlToken);

  // Retrieve attributes from the token
  List<SAMLAttribute> allAttributes =
                       ((SAMLToken)samlToken).getSAMLAttributes();

  // Iterate through the attributes and process accordingly
  Iterator<SAMLAttribute> iter = allAttributes.iterator();
  while (iter.hasNext()) {
   SAMLAttribute attribute = iter.next();
   String attributeName = attribute.getName();
   String[] attributeValues = attribute.getStringAttributeValue();
   System.out.println("attribute name = "+ attributeName +
                      "  attribute value = ["+
                      attributeValues[0]+ ",
                      "+attributeValues[1]+ ", "+
                      attributeValues[2]+"]");
  }
  } catch(Exception e) {
     e.printStackTrace();
  }
 }
}
```

Sample code output

```
default holder of key confirmation key type is: http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
authentication method for requester is: Password
confirmation method for requester is: holder-of-key
key alias for requester is: null
key alias for recipient as set in requester config is SOAPRecipient
holder of key confirmation key type is: http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
time to live from the default provider config: 3600000
keyStore path from default provider config: C:/saml/samlsample/saml-provider.jceks
keyStore type from default provider config: jceks
key alias from default provider config: samlissuer
token's creation Date is: Mon Sep 14 15:49:00 CDT 2009
token's expiration Date is: Mon Sep 14 16:49:00 CDT 2009
token's subject confirmation method is: urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
attribute name = EmployeeInfo  attribute value = [GreenRoofing, JohnDoe, 19XY981245]
```

Creating a SAML bearer token using the API:

Use the SAML library API to create a SAML bearer token.

About this task

This library allows you to create a SAML bearer token. You can use the SAML library API to create required SAML configuration objects, then use those configuration objects to generate a bearer SAML token.

Procedure

1. Create a SAMLTokenFactory instance using the SAML token version as a parameter.
   a. Use the following line of code to import the method:

```
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
```

b. Use one of these lines of code to create the instance, depending on the token version.
   - Add the following line of code to create a SAMLTokenFactory instance for a version 1.1 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

   - Add the following line of code to create a SAMLTokenFactory instance for a version 2.0 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

2. After you create the instance, the SAMLTokenFactory is used to create a RequesterConfig instance, which determines how the token will be generated, according the authentication requirements of the requester. Use this line of code to create the RequesterConfig instance for the bearer token:

```
RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();
```

   The default RequestConfig instance is sufficient to generate a simple bearer token, but additional assertions can be included in the SAML token by customizing the RequesterConfig instance. For example, to include password authentication information in the token, use setAuthenticationMethod:

```
reqData.setAuthenticationMethod("password");
```

   To disable signing in a SAML assertion, use the setAssertionSignatureRequired method, for example:

```
reqData.setAssertionSignatureRequired(false);
```

   For more information, read about the SAML bearer assertion.

3. Use the SAMLTokenFactory to create a ProviderConfig instance, which describes the token issuer. The ProviderConfig instance specifies the SAML issuer name, as well as keystore and truststore information, which identifies the key for SAML encryption and signing. The ProviderConfig instance is created using property values from a property file. The property file specifies the default value of the ProviderConfig object. In a Java client environment, this property file is defined by a JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath.

   In the WebSphere Application Server runtime environment, the property file name is SAMLIssuerConfig.properties. The file can be located either under the server level configuration directory, or the cell level directory, in that order of precedence. An example of the server level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

   An example of the cell level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

   The JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath, is ignored if the property is defined in server runtime environment. For a detailed description of all the properties, read about configuration of a SAML token during token creation.

   Use the following line of code to create a default ProviderConfig instance:

```
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any issuer name");
```

   The issuer name is optional. If the issuer name is specified, the Issuer name appears in the SAML assertion. If the issuer name is not specified, a default issuer name property from the SAMLIssuerConfig.properties is used as the issuer name.

4. Optional: When creating a new SAML token, the SAMLTokenFactory uses either a JAAS subject or a CredentialConfig instance to populate the new SAML token. To use a JAAS subject to populate the token, use the com.ibm.websphere.security.auth.WSSubject getCallerSubject() API or the getRunAsSubject() API to obtain a JAAS subject that represents the requesting client, or the identity of the execution thread.

   When you use the JAAS subject to create a new SAML token, the SAMLTokenFactory searches for a SAMLToken object in the subject PrivateCredentials list. If a SAMLToken object exists, the NameId or NameIdentifier are copied to the new SAML token. The SAMLTokenFactory also copies the SAML attributes and AuthenticationMethod from the existing SAML token to the new SAML token. The new

SAML token includes a new issuer name, new signing certificate, confirmation method, new KeyInfo for the holder-of-key confirmation method, and new NotBefore and NotOnAfter conditions. These token settings are determined by the configuration parameters in the ProviderConfig and RequesterConfig objects.

If there is no SAMLToken object in the subject, only the WSPrincipal principal name is copied from the subject to the new SAML token. No other attributes in the subject are copied into the new SAML token. Similarly, the issuer name, signing certificate, confirmation method, KeyInfo for the holder-of-key, and NotBefore and NotOnOrAfter conditions are determined by configuration parameters in ProviderConfig and RequesterConfig objects.

Alternately, you can use the RunAsSubject method on the execution thread to create the SAML token. When using this method, do not pass the JAAS subject or the CredentialConfig object to the SAMLTokenFactory to create the SAML token. Instead, the content of the existing SAML token is copied to the new SAML token, as described previously.

Another method of creating a SAML token is to use a CredentialConfig object to populate the SAML NameId and Attributes programmatically. Use this method in the following circumstances:

- Custom SAML attributes must be included in the new SAML token.
- The SAML token is created manually instead of using the SAMLTokenFactory to populate the SAML token from a JAAS subject automatically.
- There is no existing SAML token in the subject.
- There is no JAAS subject available.

To create a CredentialConfig object without using the JAAS subject, use this line of code:

```
CredentialConfig cred = samlFactory.newCredentialConfig ();
```

There is no initial value provided for this CredentialConfig object, so you must use setter methods to populate the CredentialConfig object.

To populate the SAML NameIdentifier or NameID, use the following line of code:

```
cred.setRequesterNameID("any name");
```

The value of the *any name* variable is used as the principal name in the SAML token. The name appears in the assertion as the NameIdentifier in a SAML Version 1.1 token, or NameId in the SAML Version 2.0 token. For example, if the value of *any name* is `Alice`, the following assertion is generated in a SAML Version 1.1 token:

```
<saml:NameIdentifier>Alice</saml:NameIdentifier>
```

The following assertion is generated in a SAML Version 2.0 token:

```
<saml2:NameID>Alice</saml2:NameID>
```

To include SAML attributes in the <AttributeStatement> portion of an assertion, use this code:

```
SAMLAttribute samlAttribute = new SAMLAttribute("email" /* Name*/, new String[] {"joe@websphere"}
/*Attribute Values*/, null, "IBM WebSphere namespace" /* namespace*/, "email" /* format*/, "joe" /*friendly name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(samlAttribute)
sattribute = new SAMLAttribute("Membership", new String[] {"Super users", "Gold membership"}, null, null /* format*/, null, null );
al.add(samlAttribute );
cred.setSAMLAttributes(al);
```

This sample code generates the following <Attribute> assertions:

```
<saml:Attribute AttributeName="email" NameFormat="email" AttributeNamespace="IBM WebSphere namespace">
<saml:AttributeValue>joe@websphere</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute AttributeName="Membership">
<saml:AttributeValue>Super users</saml:AttributeValue><saml:AttributeValue>Gold membership</saml:AttributeValue>
</saml:Attribute>
```

5. Generate a SAML bearer token using this line of code:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

This method requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken.

Complete code samples using lines of code from the previous steps are included in the Example section.

Example

Use this sample code to create a SAML version 1.1 bearer token from the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11)

RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("WebSphere Server");

Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 1.1 bearer token without using the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

RequesterConfig reqData = samlFactory.newBearerTokenGenerateConfig();
reqData.setAuthenticationMethod("Password"); //Authentication method for Assertion

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(Self issuer);

CredentialConfig cred = samlFactory.newCredentialConfig ();
cred.setRequesterNameID("Alice");   // SAML NameIdentifier
//SAML attributes:
SAMLAttribute attribute = new SAMLAttribute
   ("email" /* Name*/, new String[] {"joe@websphere"}
   /*Attribute Values in String*/,null
   /*Attribute Values in XML */, "WebSphere" /* Namespace*/, "email" /* format*/, "joe" /*Friendly_name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(attribute);
attribute = new SAMLAttribute("Membership", new String[] {"Super users", "My team"}, null, null, null, null  );
al.add(attribute);
cred.setSAMLAttributes(al);

SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

Creating a SAML holder-of-key token using the API:

The SAML holder-of-key token extends the security token public interface in WebSphere Application Server, and can be used as a protection token. WebSphere Application Server provides a SAML library API for SAML holder-of-key token creation.

About this task

SAML token creation requires three parameters:

- com.ibm.wsspi.wssecurity.saml.config.RequesterConfig
- com.ibm.wsspi.wssecurity.saml.config.ProviderConfig
- com.ibm.wsspi.wssecurity.saml.config.CredentialConfig

Follow the steps to create an instance for each of the parameters and then create a SAML holder-of-key token. As an alternative to CredentialConfig, you can also use javax.security.auth.Subject. For more information, read the API documentation.

Procedure

1. Create a com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory instance using the SAML token version as a parameter. The supported SAMLToken versions are http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1 and http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0. The SAMLTokenFactory instance is a singleton and therefore is thread-safe. Use one of these lines of code to create the instance, depending on the token version.
   - Use the following line of code to create a com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory instance for a version 1.1 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

- Use the following line of code to create a com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory instance for a version 2.0 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

2. The SAMLTokenFactory instance is used to create a RequesterConfig instance, which determines how the token is generated, according the authentication requirements of the requester. Use one of these lines of code to create the RequesterConfig instance, depending on whether you want the token to use a secret key (symmetric key) or a public key:
   - Use the following line of code to create a default RequesterConfig for the SAML holder-of-key token using a secret key (symmetric key), which is included in the SubjectConfirmation:

```
RequesterConfig reqData = samlFactory. newSymmetricHolderOfKeyTokenGenerateConfig ();
```

     You must also set the key alias for the target service so the provider can encrypt the secret key for the service:

```
reqData.setKeyAliasForAppliesTo("Soap Recipient");
```

   - Use the following line of code to create a default RequesterConfig for the SAML holder-of-key token using a public key, which is included in the SubjectConfirmation:

```
RequesterConfig reqData = samlFactory. newAsymmetricHolderOfKeyTokenGenerateConfig ();
```

     You must also set the key alias for the requester so the provider can extract the public key from the requester, and include the key in the SubjectConfirmation:

```
reqData.setKeyAliasForRequester("SOAP Initiator");
```

   The default RequestConfig instance is sufficient to generate a simple holder-of-key token, but additional assertions can be included in the SAML token by customizing the RequesterConfig instance. For example, to include password authentication information in the token, use setAuthenticationMethod:

```
reqData.setAuthenticationMethod("password");
```

3. Use the SAMLTokenFactory to create a ProviderConfig instance, which describes the token issuer. The ProviderConfig instance specifies the SAML issuer name, as well as keystore and truststore information, which identifies the key for SAML encryption and signing. The ProviderConfig instance is created using property values from a property file. The property file specifies the default value of the ProviderConfig object. In a Java client environment, this property file is defined by a JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath.

   In the WebSphere Application Server runtime environment, the property file name is SAMLIssuerConfig.properties. The file can be located either under the server level configuration directory, or the cell level directory, in that order of precedence. An example of the server level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

   An example of the cell level path follows:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

   The JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath, is ignored if the property is defined in server runtime environment. For a detailed description of all the properties, read about configuration of a SAML token during token creation.

   Use the following line of code to create a default ProviderConfig instance:

```
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any issuer name");
```

   The issuer name is optional. If the issuer name is specified, the Issuer name appears in the SAML assertion. If the issuer name is not specified, a default issuer name property from the SAMLIssuerConfig.properties is used as the issuer name.

4. Optional: When creating a new SAML token, the SAMLTokenFactory uses either a JAAS subject or a CredentialConfig instance to populate the new SAML token. To use a JAAS subject to populate the

token, use the com.ibm.websphere.security.auth.WSSubject getCallerSubject() API or the getRunAsSubject() API to obtain a JAAS subject that represents the requesting client, or the identity of the execution thread.

When you use the JAAS subject to create a new SAML token, the SAMLTokenFactory searches for a SAMLToken object in the subject PrivateCredentials list. If a SAMLToken object exists, the NameId or NameIdentifier are copied to the new SAML token. The SAMLTokenFactory also copies the SAML attributes and AuthenticationMethod from the existing SAML token to the new SAML token. The new SAML token includes a new issuer name, new signing certificate, confirmation method, new KeyInfo for the holder-of-key confirmation method, and new NotBefore and NotOnAfter conditions. These token settings are determined by the configuration parameters in the ProviderConfig and RequesterConfig objects.

If there is no SAMLToken object in the subject, only the WSPrincipal principal name is copied from the subject to the new SAML token. No other attributes in the subject are copied into the new SAML token. Similarly, the issuer name, signing certificate, confirmation method, KeyInfo for the holder-of-key, and NotBefore and NotOnOrAfter conditions are determined by configuration parameters in ProviderConfig and RequesterConfig objects.

Alternately, you can use the RunAsSubject method on the execution thread to create the SAML token. When using this method, do not pass the JAAS subject or the CredentialConfig object to the SAMLTokenFactory to create the SAML token. Instead, the content of the existing SAML token is copied to the new SAML token, as described previously.

Another method of creating a SAML token is to use a CredentialConfig object to populate the SAML NameId and Attributes programmatically. Use this method in the following circumstances:

- Custom SAML attributes must be included in the new SAML token.
- The SAML token is created manually instead of using the SAMLTokenFactory to populate the SAML token from a JAAS subject automatically.
- There is no existing SAML token in the subject.
- There is no JAAS subject available.

To create a CredentialConfig object without using the JAAS subject, use this line of code:

```
CredentialConfig cred = samlFactory.newCredentialConfig ();
```

There is no initial value provided for this CredentialConfig object, so you must use setter methods to populate the CredentialConfig object.

To populate the SAML NameIdentifier or NameID, use the following line of code:

```
cred.setRequesterNameID("any name");
```

The value of the *any name* variable is used as the principal name in the SAML token. The name appears in the assertion as the NameIdentifier in a SAML Version 1.1 token, or NameId in the SAML Version 2.0 token. For example, if the value of *any name* is `Alice`, the following assertion is generated in a SAML Version 1.1 token:

```
<saml:NameIdentifier>Alice</saml:NameIdentifier>
```

The following assertion is generated in a SAML Version 2.0 token:

```
<saml2:NameID>Alice</saml2:NameID>
```

To include SAML attributes in the <AttributeStatement> portion of an assertion, use this code:

```
SAMLAttribute samlAttribute = new SAMLAttribute("email" /* Name*/, new String[] {"joe@websphere"}
/*Attribute Values*/, null, "IBM WebSphere namespace" /* namespace*/, "email" /* format*/, "joe" /*friendly name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(samlAttribute)
sattribute = new SAMLAttribute("Membership", new String[] {"Super users", "Gold membership"}, null, null /* format*/, null, null );
al.add(samlAttribute );
cred.setSAMLAttributes(al);
```

This sample code generates the following <Attribute> assertions:

```
<saml:Attribute AttributeName="email" NameFormat="email" AttributeNamespace="IBM WebSphere namespace">
<saml:AttributeValue>joe@websphere</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute AttributeName="Membership">
<saml:AttributeValue>Super users</saml:AttributeValue><saml:AttributeValue>Gold membership</saml:AttributeValue>
</saml:Attribute>
```

5. Generate a SAML holder-of-key token using this line of code:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

This method requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Complete code samples using lines of code from the previous steps are included in the Example section.

Example

Use this sample code to create a SAML version 1.1 holder-of-key token using a secret key (symmetric key) from the subject.

```
import com.ibm.wsspi.wssecurity.saml.config.RequesterConfig;
import com.ibm.wsspi.wssecurity.saml.config.ProviderConfig;
import com.ibm.wsspi.wssecurity.saml.config.CredentialConfoig ;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory

SAMLTokenFactory samlFactory =  SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

RequesterConfig reqData  =  samlFactory.newSymmetricHolderOfKeyTokenGenerateConfig();

//Map "AppliesTo" to key alias, so library knows how to encrypt the Symmetric Key
reqData.setKeyAliasForAppliesTo("SOAPRecipient");

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(IsserUri);

Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 2.0 holder-of-key token using a public key from the subject:

```
//User expression on how SAML should be created, default provided
RequesterConfig reqData =  samlFactory.newAsymmetricHolderOfKeyTokenGenerateConfig();

//Choose a public key to be included in SAML
reqData.setKeyAliasForRequester("SOAPInitiator");

//Get issuer key store so can sign or encrypt assertion, issuer name
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any_issuer");

//Get JAAS Subject so the factory can populate principal and attributes to SAML
Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 2.0 holder-of-key token using a secret key (symmetric key):

```
SAMLTokenFactory samlFactory =  SAMLTokenFactory.getInstance (SAMLTokenFactory.WssSamlV20Token11);

RequesterConfig reqData = samlFactory. newSymmetricHolderOfKeyTokenGenerateConfig ();
//Map "AppliesTo" to key alias so library knows how to encrypt the Symmetric Key
reqData.setKeyAliasForAppliesTo("SOAPRecipient");

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(null);

CredentialConfig cred = samlFactory.newCredentialConfig ();
cred.setRequesterNameID("any_name");

SAMLToken samlToken =  samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Creating a SAML sender-vouches token using the API:

Use the SAML library API to create a SAML sender-vouches token, which includes the sender-vouches confirmation method. The sender-vouches confirmation method is used when a server needs to propagate the client identity or behavior of the client.

About this task

When SAML function is installed on a WebSphere server, a SAML library API is provided. Use the library to create a SAML sender-vouches token. You can use the SAML library API to create required SAML configuration objects. Then, use those configuration objects to generate a SAML sender-vouches token.

Procedure

1. Create a SAMLTokenFactory instance using the SAML token version as a parameter.
   a. Use the following line of code to import the method:

```
import com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory;
```

   b. Use one of these lines of code to create the instance, depending on the token version.
      - Add the following line of code to create a SAMLTokenFactory instance for a version 1.1 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);
```

      - Add the following line of code to create a SAMLTokenFactory instance for a version 2.0 SAML token:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token20);
```

2. After you create the instance, the SAMLTokenFactory is used to create a RequesterConfig instance, which determines how the token will be generated, according the authentication requirements of the requester. Use this line of code to create the RequesterConfig instance for the sender-vouches token:

```
RequesterConfig reqData = samlFactory.newSenderVouchesTokenGenerateConfig();
```

   The default RequestConfig instance is sufficient to generate a simple sender-vouches token, but additional assertions can be included in the SAML token by customizing the RequesterConfig instance. For example, to include password authentication information in the token, use the method, setAuthenticationMethod:

```
reqData.setAuthenticationMethod("password");
```

   The trust validation for a sender-vouches assertion is the responsibility of the sender, not the issuer, so the Enveloped-Signature element is not required in the assertion. To remove the Enveloped-Signature element from the SAML assertion, use the setAssertionSignatureRequired method; for example:

```
reqData.setAssertionSignatureRequired(false);
```

3. Use the SAMLTokenFactory API to create a ProviderConfig instance, which describes the token issuer. The ProviderConfig instance specifies the SAML issuer name, as well as keystore and truststore information, which identifies the key for SAML encryption and signing. The ProviderConfig instance is created using property values from a property file. The property file specifies the default value of the ProviderConfig object. In a Java client environment, this property file is defined by a JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath.

   In the WebSphere Application Server runtime environment, the property file name is SAMLIssuerConfig.properties. The file can be located either under the server level configuration directory, or the cell level directory, in that order of precedence. See the following example of the server-level path:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/nodes/$NODENAME/servers/$SERVERNAME/SAMLIssuerConfig.properties
```

   See the following example of the cell-level path:

```
app_server_root/profiles/$PROFILE/config/cells/$CELLNAME/sts/SAMLIssuerConfig.properties
```

   The JVM system property, com.ibm.webservices.wssecurity.platform.SAMLIssuerConfigDataPath, is ignored if the property is defined in server runtime environment. For a detailed description of all the properties, read about configuration of a SAML token during token creation.

   Use the following line of code to create a default ProviderConfig instance:

```
ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("any issuer name");
```

The issuer name is optional. If the issuer name is specified, the Issuer name appears in the SAML assertion. If the issuer name is not specified, a default issuer name property from the SAMLIssuerConfig.properties is used as the issuer name.

4. Optional: When creating a new SAML token, the SAMLTokenFactory uses either a Java Authentication and Authorization Service (JAAS) subject or a CredentialConfig instance to populate the new SAML token. To use a JAAS subject to populate the token, use the com.ibm.websphere.security.auth.WSSubject getCallerSubject() API or the getRunAsSubject() API to obtain a JAAS subject that represents the requesting client, or the identity of the execution thread.
   - When you use the JAAS subject to create a new SAML token, the SAMLTokenFactory API searches for a SAMLToken object in the subject PrivateCredentials list. If a SAMLToken object exists, the NameId or NameIdentifier objects are copied to the new SAML token. The SAMLTokenFactory also copies the SAML attributes and AuthenticationMethod method from the existing SAML token to the new SAML token. The new SAML token includes a new issuer name, new signing certificate, confirmation method, new KeyInfo for the holder-of-key confirmation method, and new NotBefore and NotOnAfter conditions. These token settings are determined by the configuration parameters in the ProviderConfig and RequesterConfig objects.

     If there is no SAMLToken object in the subject, only the WSPrincipal principal name is copied from the subject to the new SAML token. No other attributes in the subject are copied into the new SAML token. Similarly, the issuer name, signing certificate, confirmation method, KeyInfo for the holder-of-key, and NotBefore and NotOnOrAfter conditions are determined by configuration parameters in ProviderConfig and RequesterConfig objects.

     Alternately, you can use the RunAsSubject method on the execution thread to create the SAML token. When using this method, do not pass the JAAS subject or the CredentialConfig object to the SAMLTokenFactory to create the SAML token. Instead, the content of the existing SAML token is copied to the new SAML token, as described previously.
   - Another method of creating a SAML token is to use a CredentialConfig object to populate the SAML NameId and Attributes programmatically. Use this method in the following circumstances:
     – Custom SAML attributes must be included in the new SAML token.
     – The SAML token is created manually instead of using the SAMLTokenFactory to populate the SAML token from a JAAS subject automatically.
     – There is no existing SAML token in the subject.
     – There is no JAAS subject available.

   a. To create a CredentialConfig object without using the JAAS subject, use this line of code:

```
CredentialConfig cred = samlFactory.newCredentialConfig ();
```

      There is no initial value provided for this CredentialConfig object, so you must use setter methods to populate the CredentialConfig object.

   b. To populate the SAML NameIdentifier or NameID, use the following line of code:

```
cred.setRequesterNameID("any name");
```

      The value of the parameter passed to the setRequesterNameID() method is used as the principal name in the SAML token. The name appears in the assertion as the NameIdentifier in a SAML Version 1.1 token, or NameId in the SAML Version 2.0 token. For example, if the value of the parameter passed to the setRequesterNameID() method is `Alice`, the following assertion is generated in a SAML Version 1.1 token:

```
<saml:NameIdentifier>Alice</saml:NameIdentifier>
```

      The following assertion is generated in a SAML Version 2.0 token:

```
<saml2:NameID>Alice</saml2:NameID>
```

   c. To include SAML attributes in the <AttributeStatement> portion of an assertion, use this code:

```
SAMLAttribute samlAttribute = new SAMLAttribute("email" /* Name*/, new String[] {"joe@websphere"}
/*Attribute Values*/, null, "IBM WebSphere namespace" /* namespace*/, "email" /* format*/, "joe" /*friendly name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(samlAttribute)
sattribute = new SAMLAttribute("Membership", new String[] {"Super users", "Gold membership"}, null, null /* format*/, null, null );
al.add(samlAttribute );
cred.setSAMLAttributes(al);
```

This sample code generates the following <Attribute> assertions:

```
<saml:Attribute AttributeName="email" NameFormat="email" AttributeNamespace="IBM WebSphere namespace">
<saml:AttributeValue>joe@websphere</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute AttributeName="Membership">
<saml:AttributeValue>Super users</saml:AttributeValue><saml:AttributeValue>Gold membership</saml:AttributeValue>
</saml:Attribute>
```

5. Generate a SAML sender-vouches token using this line of code:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

This method requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken.

Complete code samples using lines of code from the previous steps are included in the Example section.

Example

Use this sample code to create a SAML version 1.1 sender-vouches token from the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11)

RequesterConfig reqData = samlFactory.newSenderVouchesTokenGenerateConfig();

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig("WebSphere Server");

Subject subject =  com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();

SAMLToken samlToken = samlFactory.newSAMLToken(subject, reqData, samlIssuerCfg);
```

Use this sample code to create a SAML version 1.1 sender-vouches token without using the subject:

```
SAMLTokenFactory samlFactory = SAMLTokenFactory.getInstance(SAMLTokenFactory.WssSamlV11Token11);

RequesterConfig reqData = samlFactory.newSenderVouchesTokenGenerateConfig();
reqData.setAuthenticationMethod("Password"); //Authentication method for Assertion

ProviderConfig samlIssuerCfg = samlFactory.newDefaultProviderConfig(Self issuer);

CredentialConfig cred = samlFactory.newCredentialConfig ();
cred.setRequesterNameID("Alice");   // SAML NameIdentifier
//SAML attributes:
SAMLAttribute attribute = new SAMLAttribute
   ("email" /* Name*/, new String[] {"joe@websphere"}
   /*Attribute Values in String*/,null
   /*Attribute Values in XML */, "WebSphere" /* Namespace*/, "email" /* format*/, "joe" /*Friendly_name */);
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(attribute);
attribute = new SAMLAttribute("Membership", new String[] {"Super users", "My team"}, null, null, null, null  );
al.add(attribute);
cred.setSAMLAttributes(al);

SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

Propagation of SAML tokens using the API:

The SAML propagation function is useful for applications that interact across multiple servers. The propagation feature communicates token information from the originating server downstream to other servers.

You can propagate SAML tokens using administrative commands, or programmatically using the SAML application programming interface (API). Propagation through administrative commands is discussed in the topics Propagating SAML tokens and SAML token propagation methods.

Programmatic propagation of SAML tokens is achieved through a combination of explicit programming and use of the Web Services Security runtime environment. For example, you can extract the SAMLToken from the org.apache.axis2.jaxws.BindingProvider object. The token is then used for outbound calls. In this

example, since WebSphere security is not required, programmatically propagating the SAML token allows you to exploit SAML security at the application level. Furthermore, the SAML token can be communicated downstream using any protocol.

Use the following sample code to extract the SAMLToken on the client side after the first request is completed.

Create a Dispatch object and invoke the request:

```
javax.xml.ws.Dispatch dispatch = ...;
dispatch.invoke();
```

Obtain a response context and extract the SAMLToken:

```
Map<String, Object> responseContext = dispatch.getResponseContext();
   SAMLToken samlToken =
       (SAMLToken ) responseContext.get(com.ibm.wsspi.wssecurity.saml.config.SamlConstants.
    SAMLTOKEN_OUT_MESSAGECONTEXT);
```

The following sample code shows how to reuse a SAMLToken for subsequent web services requests.

The web services client program creates a dispatch instance to invoke a service:

```
javax.xml.ws.Dispatch dispatch = ...;
```

The web services client then uses this code to pass a SAMLToken to the Web Services Security handler:

```
Map<String, Object> requestContext = dispatch.getRequestContext();
     requestContext.put(com.ibm.wsspi.wssecurity.saml.config.SamlConstants.
     SAMLTOKEN_IN_MESSAGECONTEXT, samlToken);
```

The web services provider (receiver) can use the following code to extract a SAMLToken from an incoming web services request.

Extract a SAMLToken from the requestContext:

```
Subject subject = (Subject) context.get(com.ibm.wsspi.wssecurity.core.Constants.WSSECURITY_TOKEN_WSSSUBJECT);
 SAMLToken samlToken = null;
     try
     {
      samlToken = (SAMLToken) AccessController.doPrivileged(
        new java.security.PrivilegedExceptionAction() {
         public Object run() throws
             java.lang.Exception
         {
          final java.util.Iterator authIterator =
                subject.getPrivateCredentials(SAMLToken.class)
                .iterator();
          if ( authIterator.hasNext() ) {
           final SAMLToken token = (SAMLToken)
                  authIterator.next();

           return token;
          }
          return null;
         }
        });
 } catch (Exception ex) {
  // Error handling
```

Extract the SAML attributes:

```
List<SAMLAttribute> allAttributes;
allAttributes = ((SAMLToken) samlToken).getSAMLAttributes();
```

The web services client runtime environment can cache the SAML token. On subsequent client requests within the application, the security runtime environment retrieves the SAML token from the cache for use with the target.

Web services client token cache for SAML:

When a SAML token is initially requested, the web services runtime environment automatically caches the SAMLToken. As a result of this automatic client token caching function, subsequent web services requests can use the SAMLToken from the previous request.

The web services client token cache for SAML enables web services clients to reuse SAML tokens when accessing business services. Reusing valid SAML tokens reduces traffic to the Security Token Service (STS) and also reduces the performance impact of sending WS-Trust request messages. There are several requirements for a token to be considered valid and therefore available for caching and reuse.

In order for a SAML token to be reused, the expiration time of the token must be equivalent to, or greater than, the current time. A cache cushion is added to the current time when comparing the token expiration time with the current time so that the token does not expire immediately after it is sent.

In addition, a token is valid only if it is sent again to the same business service. The SAML function in WebSphere Application Server does not verify the AudienceRestriction condition for the SAML token. Therefore, a practical way to ensure that the SAML token is reused for the right audience is to reuse the token only for the same web service that originally used the token. If an assertion contains the OneTimeUse assertion, the SAML token is not cached.

To take advantage of the SAMLToken cache, the application and SAMLToken must meet the following requirements:

- The SAMLToken must have a relatively long expiration time, with at least 5 minutes remaining in the token lifetime after the first request is completed. The WS-Security runtime environment validates the cached SAMLToken expiration time against a predefined cache cushion. The cached token is valid only if the remaining token lifetime is greater than the cushion value. The default cushion value is 5 minutes. This value can be configured using the custom property, cacheCushion. To override the default cache cushion, edit the CallbackHandler custom property for the SAMLToken generator. Add the cacheCushion property and set the cache cushion value in milliseconds. If the cached SAMLToken lifetime is within the cache cushion limit, a new SAMLToken is requested. For example, you can change the cache cushion to 3 minutes or 180000 milliseconds.

Custom property name	Value
cacheCushion	180000

- The SAML token cannot contain the OneTimeUse assertion.
- If the SAML token is encrypted, make sure that the STS communicates the token expiration time outside the encrypted token, and that the SAML token does not include the OneTimeUse assertion.

When you do not want to reuse the same SAMLToken for subsequent requests, you can disable the client side SAMLToken cache with the cacheToken custom property. To disable the client side SAMLToken cache, modify the custom property in the CallbackHandler for the SAMLToken generator. Add the cacheToken property and set the value to `false`.

Custom property name	Value
cacheToken	false

Securing web services applications using the WSS APIs at the message level

Standards and profiles address how to provide protection for messages that are exchanged in a web service environment. Web Services Security is a message-level standard that is based on securing SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens.

Before you begin

To secure web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, delegation, and auditing across a spectrum of application and business topologies. One of the key requirements for the security model in today's business environment is the ability to interoperate between formerly incompatible security technologies in heterogeneous environments. The complete Web Services Security protocol stack and technology roadmap is described in the web services roadmap.

About this task

The Organization for the Advancement of Structured Information Standards (OASIS) Web Services Security: SOAP Message Security Version 1.1 specification is the basic messaging transport for all web services. SOAP 1.2 adds extensions to the existing SOAP 1.1 extensions so that you can build secure web services. Attachments can be added to SOAP messages by using Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP) instead of the SOAP with Attachments (SWA) profile.

The OASIS Web Services Security (WS-Security) Version 1.1 specification is the building block that is used in conjunction with other web service and application-specific protocols to accommodate a wide variety of security models. Web Services Security for WebSphere Application Server is based on specific standards that are included in the OASIS Web Services Security Version 1.1 specification and profiles.

The Version 1.1 specification defines additional facilities for protecting the integrity and confidentiality of a message. The Version 1.1 specification also provides the mechanisms for associating security-related claims with the message. The Web Services Security Version 1.1 standards that are supported by WebSphere Application Server include the signature confirmation, encrypted header elements, the Username Token Profile and the X.509 Token Profile. The Username Token Profile and the X.509 Token Profile have been updated as Version 1.1 profiles. For the X.509 Certificate Token Profile, one new type of security token reference is the Thumbprint reference, which is specified in the binding.

XML Schema, Part 1 and Part 2 are specifications that explain how schemas are organized in XML documents. The two WS-Security Version 1.0 schemas have been updated to the Version 1.1 specifications plus a new Version 1.1 schema has been added. Note that the Version 1.1 schema does not replace the Version 1.0 schema but instead builds upon it by defining an additional set of capabilities within a Version 1.1 namespace.

You can use the following methods to configure Web Services Security and to define policy types to secure the SOAP messages:

- **Use the administrative console to configure policy sets.**

  This method uses the bootstrap policy that is defined in the policy set. You can use policy sets, or assertions about how services are defined, to simplify your security configuration for web services. You can use the administrative console to create, modify, and delete custom policy sets. A set of default policy sets are available.

  For example, you can define the bootstrap policy in the policy set to secure the Web Services Trust (WS-Trust) SOAP messages.

  You can also use the administrative console to perform policy set management tasks and to secure web services using encryption, signing information, and security tokens.

  The following steps high-level steps describe how to configure WebSphere Application Server to use WS-Security and to secure the SOAP messages using the administrative console. The generator and consumer tasks that are discussed in the following steps use WS-Security Versions 1.0 and 1.1.

  – Create and configure the application policy sets or the system policy sets for trust service.

  – Define the policy types to be used to secure the SOAP messages when creating and configuring the policy sets.

- Configure the policy set binding. Select either the symmetric or asymmetric binding assertion to describe the token type and the algorithm to be used for message protection.
- Assemble your Web Services Security-enabled application by using an assembly tool.

- **Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)**

  WebSphere Application Server uses a new API programming model. In addition to the existing JAX-RPC programming model, a new programming model, Java API for XML Web Services (JAX-WS), has been added. The JAX-WS programming standard aligns with the document-centric messaging model and replaces the remote procedure call programming model defined by the Java API for XML-based RPC (JAX-RPC) specification.

  For example, an application could create system policy sets and then use the WebSphere Application Server WSS API to acquire the security context token for programmatic API-based Web Services Secure Conversation (WS-SecureConversation).

  You can also use the administrative console to perform the encryption, signing, and token configuration tasks that the WSS APIs perform to secure web services.

  The following high-level steps describe how to configure WebSphere Application Server to use WS-Security and to secure the SOAP messages using the WSS APIs. The generator and consumer tasks that are discussed in the following steps use WS-Security Versions 1.0 and 1.1.

  - Use the WSSSignature API to configure the signing information for the request generator (client side) binding.

    Different message parts can be specified in the message protection for a request on the generator side. The default required parts are BODY, ADDRESSING_HEADERS and TIMESTAMP.

    The WSSSignature API also specifies the different algorithm methods to be used with the signature for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.
  - Use the WSSSignPart API if you want to change the digest method and the transform method.

    The default signed parts are WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS and WSSSignature.TIMESTAMP.

    The WSSSignPart API also specifies the different algorithm methods to be used if you added or changed the signed parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N. For example, use the WSSSignPart API if you want to generate the signature for the SOAP message using the SHA256 digest method instead of the default value of SHA1.
  - Use the WSSEncryption API to configure the encryption information on the request generator side.

    The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The default targets of encryption are BODY_CONTENT and SIGNATURE.

    The WSSEncryption API also specifies the different algorithm methods to be used to protect message confidentiality. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.
  - Use the WSSEncryptPart API if you want to set the transform method only.

    For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

    No algorithm methods are required for encrypted parts.
  - Use the WSS API to configure the token on the generator side.

    The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for creating the security token on the generator side. Different stand-alone tokens can be sent in request and response. The default token is the X509Token. The other token that can be used for signing is the DerivedKeyToken, which is used only with Web Services Secure Conversation (WS-SecureConversation).
  - Use the WSSVerification API to verify the signature for the response consumer (client side) binding.

Different message parts can be specified in the message protection for a response on the consumer side. The required targets for verification are BODY, ADDRESSING_HEADERS and TIMESTAMP.

The WSSVerification API also specifies the different algorithm methods to be used for verifying the signature and for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.

– Use the WSSVerifyPart API to change the digest method and the transform method. The required verify parts are WSSVerification.BODY, WSSVerification.ADDRESSING_HEADERS and WSSVerification.TIMESTAMP.

  The WSSVerifyPart API also specifies the different algorithm methods to be used if you added or changed the verification parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N.

– Use the WSSDecryption API to configure the decryption information for the response consumer (client side) binding.

  The decryption information on the consumer side is used for decrypting an incoming SOAP message. The targets of decryption are BODY_CONTENT and SIGNATURE. The default key encryption method is KW_RSA_OAEP.

  No algorithm methods are required for decryption.

– Use the WSSDecryptPart API if you want to set the transform method only.

  For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

  No algorithm methods are required for decrypted parts.

– Use the WSS API to configure the token on the consumer side.

  The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for validating (authenticating) the security token on the consumer side. Different stand-alone tokens can be sent in request or response.

  The WSS API adds the information for the candidate token that is used for decryption. The default token is X509Token.

- **Use the wsadmin administrative scripting tool to configure policy sets.**

  This method allows you to create, manage, and delete policy sets from the command-line or to create scripts to automate your tasks. You can use the wsadmin tool and the PolicySetManagement command group to manage default policy sets, create custom policy sets, configure policies, and manage attachments and bindings. For more information, use the policy set scripting topics in the information center.

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. You must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

Results

After completing these high-level steps for WebSphere Application Server, you have secured web services by configuring policy sets and by using the WSS API to configure encryption and decryption, the signature and signature verification information, and the consumer and generator tokens.

Securing messages at the request generator using WSS APIs:

You can secure SOAP messages by configuring signing information, encryption, and generator tokens to protect message integrity, confidentiality, and authenticity, respectively. This request (client-side) generator configuration defines the Web Services Security requirements for the outgoing SOAP message request.

Before you begin

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. Therefore, in addition to securing messages at the request generator level, you must also secure messages at the response consumer level.

About this task

The request (client-side) generator configuration requirements involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches security tokens.

To secure web service applications, you must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

You can use the following interfaces to configure Web Services Security and to define policy types to secure the SOAP messages:

- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:

Procedure

- Configure generator signing to protect message integrity.
- Configure encryption to protect message confidentiality.
- Attach generator tokens to protect message authenticity.
- Propagate self-issued SAML bearer tokens using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with message protection using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with transport protection using WSS APIs.
- “Sending self-issued SAML holder-of-key tokens with symmetric key using WSS APIs” on page 1598.
- “Sending self-issued SAML holder-of-key tokens with asymmetric key using WSS APIs” on page 1600.

Results

After completing these procedures, you have secured messages at the request generator level.

What to do next

Next, if not already configured, secure messages with signature verification, decryption, and consumer tokens at the response consumer (client-side) level.

Configuring encryption to protect message confidentiality using the WSS APIs:

You can configure encryption information for the client-side request generator (sender) bindings. Encryption information is used to specify how the generators (senders) encrypt outgoing SOAP messages. To configure encryption, specify which message parts to encrypt and specify which algorithm methods and security tokens are to be used for encryption.

Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. Prior to

configuring encryption, familiarize yourself with XML encryption.

About this task

For encryption, you must specify the following:

- Which parts of the message are to be encrypted.
- Which encryption algorithms to specify.

To configure encryption and encrypted parts on the client side, use the WSSEncryption and WSSEncryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses encryption information for the default generator to encrypt parts of the SOAP message. The WSSEncryption API configures the following required parts as encrypted parts.

Table 149. Required encrypted parts. Use encrypted parts to increase the confidentiality of SOAP messages.

Encryption parts	Description
Keywords	Keywords are used to add the encrypted parts to the SOAP message.
XPath expression	An XPath expression is used to add the encrypted parts to the SOAP message.
WSSEncryptPart object	This object adds the encrypted parts to the SOAP message.
WSSSignature object	This object adds the signature component as an encrypted part.
Header	This part adds the header in the SOAP header, specified by QName, as an encryption part.
Security token object	This object adds the security token as an encryption part.

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the message that are to be encrypted. WebSphere Application Server supports the use of the following keywords:

Table 150. Supported encryption keywords. Use keywords to specify encrypted parts.

Keyword	References
BODY_CONTENT	The keyword for the contents of the SOAP message body as an encryption target.
SIGNATURE	The keyword for the signature component as an encryption target.

If configuring using the WSS APIs, the WSSEncryption and WSSEncryptPart APIs complete these high-level steps:

Procedure

1. Use the WSSEncryption API to configure encryption. The WSSEncryption API performs these tasks by default:
   a. Generates the callback handler.
   b. Generates the generator security token object.
   c. Adds the security token reference type.
   d. Adds the signature component.
   e. Adds the WSSEncryptPart object.
   f. Adds the parts to be encrypted. Adds the default parts as targets of encryption by using keywords and XPath expressions.
   g. Adds the header in the SOAP message, specified by QName.

h. Sets the default data encryption method.
i. Specifies whether the key is to be encrypted using a Boolean value.
j. Sets the default key encryption method.
k. Selects a part reference.
l. Sets the MTOM optimization Boolean value.

2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the key (false).
   - Change the security token type from default of X.509 token.
   - Change the security token reference type from the default value of SecurityToken.REF_STR.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.
   - Turn MTOM optimization on (true).

Results

The encryption information is configured for the generator binding.

Example

The following is an example of the WSSEncryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSGenerationContext gencont = factory.newWSSGenerationContext();

X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);
WSSEncryption enc = factory.newWSSEncryption(token);

gencont.add(enc);
```

What to do next

You must configure similar decryption information for the client-side response consumer (receiver) bindings, if you have not already configured the information.

Next, review the WSSEncryption API process.

Encrypting the SOAP message using the WSSEncryption API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for request encryption on the generator side, use the WSSEncryption API to encrypt the SOAP message. The WSSEncryption API specifies which request SOAP message parts to encrypt when configuring the client.

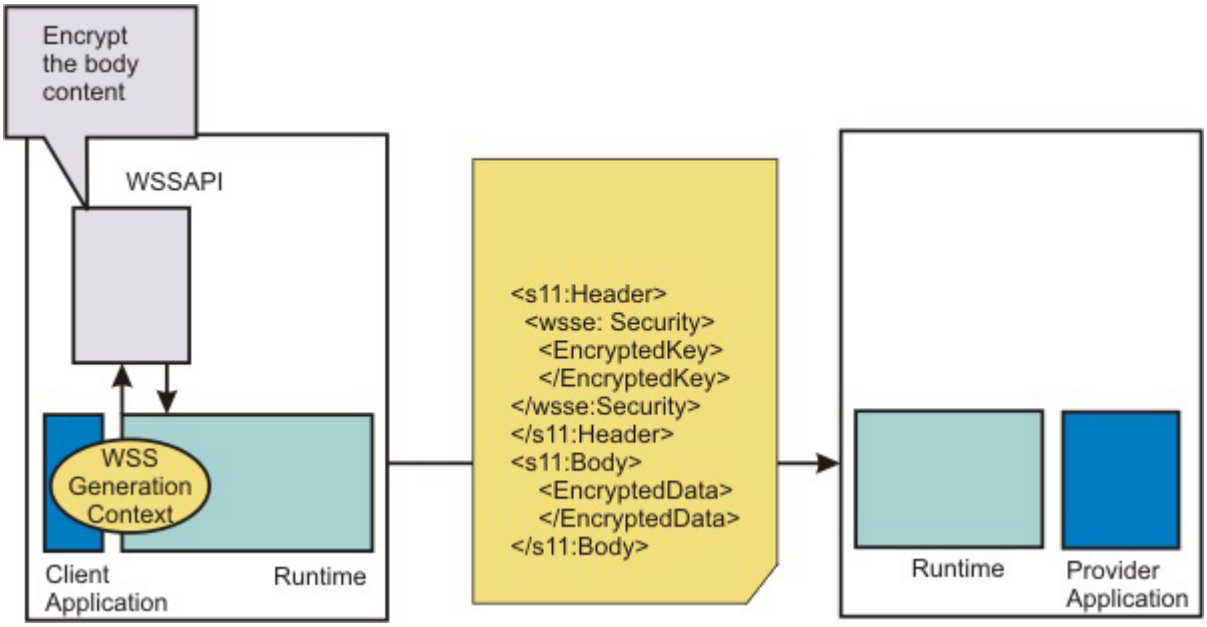
Before you begin

You can use the WSS API or use policy sets on the administrative console to enable encryption and add generator security tokens in the SOAP message. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts, as needed, using the WSSEncryptPart API.

About this task

The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The client generator configuration must match the configuration for the provider consumer.



Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted Nonce and timestamp elements to.

The following encryption parts can be configured:

Table 151. Encryption parts. Use the encryption parts to enable encryption in messages.

Encryption parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encryption parts using keywords. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encryption part using an XPath expression.
signature	Adds the WSSignature component as a target of the encrypted part.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encryption, certain default behaviors occur. The simplest way to use the WSSEncryption API is to use the default behavior (see the example code).

WSSEncryption provides defaults for the key encryption algorithm, the data encryption algorithm, the security token reference method, and the encryption parts such as the SOAP body content and the signature. The encryption default behaviors include:

Table 152. Encryption decisions. Use encryption default behavior to secure the message body content and signature.

Encryption decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Sets the encryption parts that you can add using keywords. The default encryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE
Which data encryption method to choose (algorithm)	Sets the data encryption method. Both data and key encryption methods can be specified. The default data encryption algorithm method is AES 128. WebSphere Application Server supports these data encryption methods: • WSSEncryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc • WSSEncryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc • WSSEncryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc • WSSEncryption.TRIPLE_DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc
Whether to encrypt the key (isEncrypt)	Specifies whether to encrypt the key. The values are `true` or `false`. The default value is to encrypt the key (`true`).
Which key encryption method to choose (algorithm)	Sets the key encryption method. Both data and key encryption methods can be specified. The default key encryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods: • WSSEncryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128 • WSSEncryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192 • WSSEncryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256 • WSSEncryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p • WSSEncryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5 • WSSEncryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes
Which security token to specify (securityToken)	Sets the SecurityToken. The default security token type is the X509Token. WebSphere Application Server provides the following pre-configured consumer token types: • Derived key token • X.509 tokens
Which token reference to use (refType)	Sets the type of the security token reference. The default token reference is SecurityToken.REF_KEYID. WebSphere Application Server supports the following token reference types: • SecurityToken.REF_KEYID • SecurityToken.REF_STR • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT
Whether to use MTOM (mtomOptimize)	Sets Message Transmission Optimization Mechanism (MTOM) optimization for the encrypted part.

Procedure

1. To encrypt the SOAP message using the WSSEncryption API, first ensure that the application server is installed.
2. The WSS API process for encryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory used for encryption.
   d. Creates WSSEncryption from the WSSFactory instance using the SecurityToken. The default behavior of WSSEncryption is to encrypt the body content and the signature.
   e. Adds a new part to be encrypted in WSSEncryption if the existing part is not appropriate. After addEncryptPart(), addEncryptHeader(), or addEncryptPartByXPath() is called, the default part is cleared.
   f. Calls the encryptKey(false) if the key is not to be encrypted.
   g. Sets the data encryption method if the default method is not appropriate.

h. Sets the key encryption method if the default method is not appropriate.
i. Sets the token reference if the default token reference is not appropriate.
j. Adds WSSEncryption to WSSConsumingContext.
k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample code using methods that are defined in WSSEncryption:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance (step: b)
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
X509GenerateCallbackHandler callbackHandler = new
        X509GenerateCallbackHandler(
     "",
     "enc-sender.jceks",
     "jceks",
     "storepass".toCharArray(),
     "bob",
     null,
     "CN=Bob, O=IBM, C=US",
     null);

// Generate the security token used for encryption (step: c)
SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance (step: d)
WSSEncryption enc = factory.newWSSEncryption(token);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
   enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
   enc.addEncryptHeader(new QName("http://www.w3.org/2005/08/addressing",
                                  "MessageID"));

// Set the part specified by WSSSignature (step: e)
   SecurityToken sigToken = getSecurityToken();
      WSSSignature sig = factory.newWSSSignature(sigToken);
   enc.addEncryptPart(sig);

// Set the part specified by SecurityToken (step: e)
   UNTGenerateCallbackHandler untCallbackHandler =
          new UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
                                                   untCallbackHandler);
   enc.addEncryptPart(unt, false);

// sSt the part specified by XPath expression (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
   enc.addEncryptPartByXPath(sb.toString());

// Set whether the key is encrypted (step: f)
// DEFAULT: true
   enc.encryptKey(true);

// Set the data encryption method (step: g)
```

```
// DEFAULT: WSSEncryption.AES128
   enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method (step: h)
// DEFAULT: WSSEncryption.KW_RSA_OAEP
   enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Set the token reference (step: i)
// DEFAULT: SecurityToken.REF_KEYID
  enc.setTokenReference(SecurityToken.REF_STR);

// Add the WSSEncryption to the WSSGenerationContext (step: j)
  gencont.add(enc);

// Process the WS-Security header (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler does not need the key password because the public key is used for encryption. You do not need a password to obtain the public key from the Java keystore.

What to do next

If you have not previously specified which encryption methods to choose, use the WSS API or configure the policy sets using the administrative console to choose the data and key encryption algorithm methods.

Choosing encryption methods for generator bindings:

To configure the client for request encryption for the generator binding, you must specify which encryption methods to use when the client encrypts the SOAP messages.

Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To specify which algorithm methods are to be used when the client encrypts the SOAP messages, complete the following tasks:

- Use the WSSEncryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSEncryptPart API to configure a transform algorithm method, if needed. The default is no transform algorithm.

About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

Data encryption method algorithm

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key encryption method algorithm

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the method to encrypt the key that is used to encrypt data. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property.

The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Table 153. Encryption usage types. The encryption usage types describe encryptions methods.

Usage types	Description
Data encryption	Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks.
Key encryption	Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

Data encryption

WebSphere Application Server supports the following pre-configured data encryption algorithms:

Table 154. Data encryption algorithms. These pre-configuring encryption algorithms are supported by WebSphere Application Server.

Data encryption name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, 3DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc

Key encryption

WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 155. Key encryption algorithms. These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128

Table 155. Key encryption algorithms (continued). These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	http://www.w3.org/2001/04/xmlenc#kw-tripledes

To configure the encryption and encrypted part algorithm methods, use the WSSEncryption API, or configure policy sets using the administrative console.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints that use policy sets.

The WSS API process completes the following high-level steps to specify which encryption methods to use when configuring the client for request encryption:

Procedure

1. Using the WSSEncryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The client generator configuration must match the configuration for the provider consumer.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data encryption algorithms:

   - AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - TRIPLEDES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
// Default data encryption algorithm: AES128
WSSEncryption enc = factory.newWSSEncryption(x509t);
   enc.setEncryptionMethod(EncryptionMethod.TRIPLEDES_CBC);
gencont.add(enc);
```

3. Using the WSSEncryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP

message. If the encryption key, which is the key that is used for encrypting the message parts, is not encrypted, then the decryption API selects `false` to match the encryption key.

The client generator configuration must match the configuration for the provider consumer.

The default key encryption algorithm value is key wrap RSA OAP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

- KW AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
- KW AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

  To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

  Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).KW AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

  To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
- KW RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

  The KW RSA OAEP algorithm is the default key algorithm method.

  When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
- KW RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
- KW TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

Note: For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
- algorithmName
- keyLength

Results

If there is an error condition, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample WSS API code using WSSEncryption.setEncryptionMethod() and WSSEncryption.setKeyEncryptionMethod().

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
                                  "",
                                  "enc-sender.jceks",
                                  "jceks",
                                  "storepass".toCharArray(),
                                  "bob",
```

```
                                    null,
                                    "CN=Bob, O=IBM, C=US",
                                    null);

// Generate the security token used for encryption
   SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance
   WSSEncryption enc = factory.newWSSEncryption(token);

// Set the data encryption method
// DEFAULT: WSSEncryption.AES128
      enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
      enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Add the WSSEncryption to the WSSGenerationContext
   gencont.add(enc);

// Generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

Next, if you want to add a transform algorithm, review the WSSEncryptPart API process task.

Encryption methods:

For request generator binding settings, the encryption methods include specifying the data and key encryption algorithms to use to encrypt the SOAP message. The WSS API for encryption (WSSEncryption) specifies the algorithm name and the matching algorithm uniform resource identifier (URI) for the data and key encryption methods. If the data and key encryption algorithms are specified, only elements that are encrypted with those algorithms are accepted.

Data encryption algorithms

The data encryption algorithm is used to encrypt parts of the SOAP message, including the body and the signature. Data encryption algorithms specify the algorithm uniform resource identifier (URI) for each type of data encryption algorithms.

The following pre-configured data encryption algorithms are supported:

Table 156. Data encryption algorithms. The algorithms are used to encrypt SOAP messages.

Data encryption algorithm name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For the AES256-cbc and the AES192-CBC algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm configured for encryption for the generator side must match the data encryption algorithm that is configured for decryption for the consumer side.

Key encryption algorithms

This algorithm is used to encrypt and decrypt keys. This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints using the policy sets.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The following pre-configured key encryption algorithms are supported:

Table 157. Supported pre-configured key encryption algorithms. The algorithms are used to encrypt and decrypt keys.

WSS API	URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	A URI of key encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

For Secure Conversation, additional key-related information must be specified, such as:

- algorithmName
- keyLength

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator must match the key decryption algorithm that is configured for the consumer.

This example provides sample code for encryption to use the Triple DES for the data encryption method and to use RSA1.5 for the key encryption method:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "enc-sender.jceks",
  "jceks",
  "storepass".toCharArray(),
  "bob",
  null,
  "CN=Bob, O=IBM, C=US",
  null);

// generate the security token used to the encryption
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

// generate WSSEncryption instance to encrypt the SOAP body content
WSSEncryption enc = factory.newWSSEncryption(token);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// set the data encryption method
// DEFAULT: WSSEncryption.AES128
enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// add the WSSEncryption to the WSSGenerationContext
gencont.add(enc);

// generate the WS-Security header
gencont.process(msgcontext);
```

Adding encrypted parts using the WSSEncryptPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure encrypted parts for the request generator (client side) bindings, use the WSSEncryptPart API to define and add to the listing of elements in the encrypted part. WSSEncryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.encryption package.

Before you begin

You can use the WSS APIs or configure policy sets using the administrative console to enable the encrypted parts. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts using the WSSEncryptpart API, as needed.

About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted elements to. The encryption information on the generator side is used for encrypting an outgoing SOAP message. The request generator is configured for the client.

The WSSEncryptPart API specifies information related to encrypted parts and sets the encrypted parts that have been added for message confidentiality protection. Use the WSSEncryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSEncryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The encrypted parts and related information displayed in the following table are used to protect the confidentiality of messages.

Table 158. Encrypted parts. Use encrypted parts to secure messages.

Encrypted parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encrypted parts using keywords. The default encryption parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encrypted part by using an XPath expression.
signature	Adds the WSSSignature component as a target of the encrypted part. WSSSignature is applicable only if the SOAP message contains a signature element.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encrypted parts, certain default behaviors occur. The simplest way to use the WSSEncryptPart API is to use the default behavior. The WSSEncryptPart API provides defaults for specifying the transform algorithm, setting objects as targets, specifying the encrypted parts, such as: the SOAP body content and the signature.

The encryption default behaviors include:

Table 159. Encrypted part decisions. Several encrypted message parts are set by default.

Encrypted part decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Specifies which keywords to use for the encrypted parts. WebSphere Application Server sets the following SOAP message parts by default for encryption: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE
Which transform method to add	WebSphere Application Server does not specify any transform method by default. Specify a transform method only if using SOAP with Attachments.

Procedure

1. To encrypt the SOAP message parts using the WSSEncryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSEncryptPart follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the encryption.
   d. Creates WSSEncryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSEncryptPart from WSSFactory.

f. Adds the parts to be encrypted and to be applied with the transform in WSSEncryptPart. WebSphere Application Server sets these encrypted parts by default for WSSEncryptPart: the BODY_CONTENT and SIGNATURE. After you add other encrypted parts, the default values are no longer valid. For example, if you call addEncryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to encrypt the security token, the signature, and the body content, you must call addEncryptPart(securityToken, false), addEncryptPart(WSSEncryption.SIGNATURE), and addEncryptPart(WSSEncrypyion.BODY_CONTENT).

g. Sets the transform method.

h. Adds WSSEncryptPart to WSSEncryption.

i. Adds WSSEncryption to WSSGenerationContext.

j. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption of the message parts, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

What to do next

After enabling encrypted parts for the request generator (client side) binding, you must specify the same parts to be decrypted for the response consumer (client side) bindings. Next, to configure decryption and decrypted parts, use the WSS APIs or configure policy sets using the administrative console.

Configuring generator signing information to protect message integrity using the WSS APIs:

You can configure the signing information to protect message integrity for the request (client side) generator binding. Signing information includes the signature and the signed parts. To keep the integrity of the message, digital signatures are typically applied.

Before you begin

In addition to using a digital signature and configuring the signing information, the following tasks should also be performed:

- Verify the signing information.
- Incorporate encryption.
- Attach security tokens.

About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using the signature algorithm methods. The WSS APIs specify which algorithm is to be used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports several pre-configured request signing algorithm methods.

You can use the following interfaces to configure Web Services Security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for the signing information.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client).

Perform the following signing tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the generator binding.

Procedure

- Configure the signing information using the WSSSignature API. Configure the signing information for the generator binding using the WSSSignature API. Signing information is used to sign parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both signing and encryption can be applied to the same message parts, such as the SOAP body.
- Add or change signed parts using the WSSSignPart API.
- Configure the client for request signing methods using the WSSSignature or WSSSignPart APIs. To configure the client for request signing, choose the signing methods. The request signing methods include the signature, the canonicalization, the digest, and the transform methods. Use the WSSSignature API to configure the signature and canonicalization methods. Use the WSSSignPart API to configure the digest and transform methods.

Results

The WSS APIs also specify the security token for the generator (client) binding and set the type of token reference to protect message authenticity. By completing the steps in these tasks, you have configured generator signing to protect the integrity of the SOAP message.

What to do next

Next, verify the consumer signing information by using the WSS APIs or by configuring policy sets using the administrative console.

Configuring signing information using the WSS APIs:

You can configure the signing information for the client-side request generator (sender) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token. To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method. Prior to completing these steps, familiarize yourself with XML digital signature for signing and verifying digital signatures for digital content.

About this task

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. WebSphere Application Server uses the signing information for the default generator to sign parts of the message, such as the body, time stamp, and Username token.

For the signing information, you must specify the following:

- Which parts of the message are to be signed.
- The key information that is referenced by the key information for the signing keys.

- The signing algorithms.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSSignature API configures the following parts as signature parts:

Table 160. Pre-configured signature parts. Use the signing information to validate parts of a message.

Part	Description
Security token object	This object authenticates the client. If this option is specified, then the message is signed. You can digitally sign the message using a security token if a login configuration authentication method is selected.
WSSTimestamp object	This object adds a time stamp to a message. The time stamp determines if the message is valid based on the time that the message is sent and then received.
WSSSignature Part object	This object adds the signature parts to a message.
SOAP header and the QName as a target	This signature part adds the header, specified by QName, as a verification part.

The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be signed. WebSphere Application Server supports the use of the following keywords:

Table 161. Supported signature keywords. Key information is used to specify which parts of a message are signed.

Keyword	References
ADDRESSING_HEADERS	The Web Services Addressing (WS-Addressing) headers.
BODY	The SOAP message body. The body is the user data portion of the message.
TIMESTAMP	The creation and expiration timestamp information.

The Web Services Security API (WSS API) are used to configure the signing information for the request generator (client side) section of the bindings file. To configure the signing information on the client side, use the WSS APIs or configure policy sets for signing using the administrative console.

If configuring using the WSS APIs, the WSSSignature and WSSSignPart APIs complete the following steps to specify which message parts to digitally sign when configuring the client for request generator signing:

Procedure

1. The WSSSignature API adds the required parts of the SOAP message to digitally sign. Either a keyword or an XPath expression can be used to specify the required encryption parts.
2. The WSSSignature API sets the signature method algorithm. The default signature method is RSA_SHA1. WebSphere Application Server supports the following pre-configured algorithms:
   - RSA SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - HMAC SHA1http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).

   Any ds:SignatureMethod/@Algorithm element in a signature is based on a symmetric key and must have a value of RSA-SHA1 or HMAC-SHA1.

   The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.
3. The WSSSignature API sets the canonicalization method. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured algorithms:
   - The URI of the exclusive canonicalization algorithm, EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#.
   - The URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#.

The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.

4. The WSSSignature API adds a security token. The API adds information about the security token that is to be used for the signature, such as:
   - The class for security token.
   - The callback handler
   - The name of the JAAS login configuration.
5. The WSSSignature API sets the type of security token and sets the type of token reference. WebSphere Application Server supports the following pre-configured token references:
   - SecurityToken.REF_STR

     Represents the security token reference as a token reference type.
   - SecurityToken.REF_KEYID

     Represents the key identifier reference as a token reference type.
   - SecurityToken.REF_EMBEDDED

     Represents the embedded reference as a token reference type.
   - SecurityToken.REF_THUMBPRINT

     Represents the thumbprint reference as a token reference type.
6. If SecurityToken.REF_KEYID is set as the type of token reference, the WSSSignature API sets the key information signature type and configures the key information that is referenced by the key information references. WebSphere Application Server supports the following:
   - Specifying that the KeyInfo element is not signed.
   - Specifying that the entire <KeyInfo> element is signed.
   - Specifying that the child elements <Keyinfochildelements> of the <KeyInfo> element are signed.

   If you do not specify one of the previous signature types, WebSphere Application Server specifies that the entire <KeyInfo> element is signed, by default.

   If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.

   The key information signature type for the generator must match the signature type for the consumer.
7. The WSSSignature API specifies whether to require signature confirmation. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

   The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.
8. The WSSSignPart API specifies the part reference. The part reference specifies which parts of the message to digitally sign.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature verification: <DigestTransform> and <Transform>.
9. The WSSSignPart API specifies the digest method algorithm. The digest method algorithm specified within the <DigestMethod> element is used in the <SigningInfo> element.

   WebSphere Application Server supports the following pre-configured digest algorithms:
   - http://www.w3.org/2000/09/xmldsig#sha1
   - http://www.w3.org/2001/04/xmlenc#sha256
   - http://www.w3.org/2001/04/xmlenc#sha512

10. The WSSSignPart API specifies the transform algorithm. The transform algorithm is that is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms:
    - http://www.w3.org/2001/10/xml-exc-c14n#
    - http://www.w3.org/TR/1999/REC-xpath-19991116

      Do not use this transform algorithm if you want to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.
    - http://www.w3.org/2002/06/xmldsig-filter2
    - http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
    - http://www.w3.org/2002/07/decrypt#XML
    - http://www.w3.org/2000/09/xmldsig#enveloped-signature

    The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

    Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:
    - You previously selected the Keyinfo or the Keyinfochildelements option
    - You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.
11. If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. Configure policy sets using the administrative console to configure the same actor strings for the web service on the server, which processes the request and sends the response back.

    The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

Results

After the WSSSignature and WSSSignPart APIs complete these steps, the signing information is configured for the generator sections of the bindings files.

Example

The following example shows WSS API sample code to configure the signature, to generate the callback handler, and to specify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
// Instantiate a generation context
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler and specify the X.509 token
   X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
   SecurityToken token = factory.newSecurityToken(X509Token.class,
                                                  callbackHandler);

// Set the signature information
WSSSignature sig = factory.newWSSSignature(token);
// Add the header using QName
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "To"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));
```

```
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "Action"));
// Apply the signature
   gencont.add(sig);

// Secure the message
gencont.process(msgctx);
```

What to do next

You must configure similar signature information for the client-side request consumer (receiver) bindings by completing the following verification tasks:

- Verify the signature
- Choose the signature algorithm methods.
- Change or add signed parts, as needed.

If signature verification is already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

Configuring signing information using the WSSSignature API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the signing information for the generator binding sections for the client-side request, use the WSSSignature API. The WSSSignature API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to enable the signing information. To secure SOAP messages, you must complete the following signing tasks:

- Configure the signing information.
- Choose the signing methods.
- Add or change signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-SHA1.

XML signature defines many methods for describing key information and enables the definition of a new method. XML canonicalization (C14N) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The signing information specifies the integrity constraints that are applied to generated messages. The constraints include specifying which message parts within the generated message must be digitally signed, and the message parts to attach digitally signed Nonce and timestamp elements to. The following signature and related signature part information are configured:

Table 162. Signature parts information. Use the signature parts to secure messages.

signature parts	Description
keyword	Adds a signature part using keywords. Use the following keywords for the signature parts: • ADDRESSING_HEADERS • BODY • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds a signature part by using an XPath expression.
part	Adds a WSSSignPart object as a target of the signature part.
timestamp	Adds a WSSTimestamp object as a target of the signature part. When specified, the timestamp information also specifies when the message is generated and when it expires.
header	Adds the header, specified by QName, as a target of the signature part.
securityToken	Adds a SecurityToken object as a target of the signature part.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignature API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing method, the canonicalization method, the security token references, and the signature parts.

Table 163. Signature default behaviors. Several signature behaviors are configured by default.

Signature decisions	Default behavior
Which keywords to use	Sets the keywords. WebSphere Application Server supports the following keywords by default: • ADDRESSING_HEADERS • BODY • TIMESTAMP
Which signature method to use	Sets the signature algorithm. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods: • WSSSignature.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1 • WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1 The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
Which canonicalization method to use	Sets the canonicalization algorithm. The default canonicalization method is EXC C14N. WebSphere Application Server supports the following pre-configured canonicalization methods: • WSSSignature.EXC_C14N; http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignature.C14N: http://www.w3.org/2001/10/xml-c14n#
Whether signature confirmation is required	Sets whether to require signature confirmation. The default value is **false**. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. If required, the value of your signature confirmation is stored in order to use it to validate the signature confirmation after receiving back the message that generated the signature confirmation in the response message. This method is for the requestor side.
Which security token to use	Sets the SecurityToken. The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type. WebSphere Application Server provides the following pre-configured consumer token types: • Derived Key Token • X509 tokens You can also create custom token types, as needed.
Which token reference to set	Sets the refType. SecurityToken.REF_STR is the default value for the type of token reference. WebSphere Application Server supports these pre-configured token references types: • SecurityToken.REF_STR • SecurityToken.REF_KEYID • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT

If WSSSignature.requireSignatureConfirmation() is called, then the WSSSignature API expects that the response message will include the signature confirmation.

Procedure

1. To configure the signing information in a SOAP message by using the WSS API, first ensure that the application server is installed.
2. Use the WSSSignature API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance. WSSGenerationContext must be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken. The default behavior of WSSSignature is to sign these signature parts: BODY, ADDRESSING_HEADERS, and TIMESTAMP.
   e. Adds the part to be signed, if the default part is not appropriate. If the digest method or transform method is changed, creates WSSSignPart and add it to WSSSignature.
   f. Creates WSSSignaturePart to WSSSignature. Calls the requiredSignatureConfirmation() method, if the signature confirmation is to be applied.
   g. Sets the canonicalization method, if the default is not appropriate.
   h. Sets the signature method, if the default is not appropriate.
   i. Sets the token reference, if the default is not appropriate.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signature for the generator section of the bindings. If there is an error condition when signing the message parts, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses methods that are defined in the WSSignature API.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the com.ibm.websphere.wssecurity.wssapi.WSSFactory instance  (step: a)
   WSSFactory factory = com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();

// Generate the WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
       "",
       "dsig-sender.ks",
       "jks",
       "client".toCharArray(),
       "soaprequester",
       "client".toCharArray(),
       "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token to be used for the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class,
        callbackHandler);

// Generate the WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part to be signed  (step: e)
// DEFAULT: WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS,
//          and WSSSignature.TIMESTAMP.

// Set the part in the SOAP Header specified by QName  (step: e)
      sig.addSignHeader(new
                        QName("http://www.w3.org/2005/08/addressing",
```

```
                        "MessageID"));

// Set the part specified by the keyword  (step: e)
      sig.addSignPart(WSSSignature.BODY);

// Set the part specified by SecurityToken  (step: e)
   UNTGenerateCallbackHandler untCallbackHandler = new
      UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
         untCallbackHandler);
      sig.addSignPart(unt);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();
      sigPart.setSignPart(WSSSignature.TIMESTAMP);
      sigPart.setDigestMethod(WSSSignPart.SHA256);
      sig.addSignPart(sigPart);

// Set the part specified by WSSTimestamp  (step: e)
   WSSTimestamp timestamp = factory.newWSSTimestamp();
      sig.addSignPart(timestamp);

// Set the part specified by XPath expression  (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
      sig.addSignPartByXPath(sb.toString());

// Set to apply the signature confirmation  (step: f)
      sig.requireSignatureConfirmation();

// Set the canonicalization method  (step: g)
// DEFAULT: WSSSignature.EXC_C14N
      sig.setCanonicalizationMethod(WSSSignature.C14N);

// Set the signature method  (step: h)
// DEFAULT: WSSSignature.RSA_SHA1
      sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

// Set the token reference  (step: i)
// DEFAULT: SecurityToken.REF_STR
      sig.setTokenReference(SecurityToken.REF_KEYID);

// Add the WSSSignature to WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgctx);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Next, chose the algorithm methods if you want a method that is different from the default values. If the algorithm methods do not need to be changed, next use the WSSVerification API to verify the signature and specify the algorithm methods in the consumer section of the binding. Note that the WSSVerification API is only supported on the response consumer (client side).

Adding signed parts using the WSSSignPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure parts to be signed for the request generator (client side) bindings, use the WSSSignPart API to protect the integrity of messages and to configure the digest and transform algorithm methods. The WSSSignPart API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to configure the signing information. To secure SOAP messages using the signing information, you must complete one of the following tasks:

- Configure the signature information
- Configure signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing digest and transform algorithms (for example, SHA1 or TRANSFORM_EXC_C14N).

The signing information specifies the integrity constraints that are applied to generated messages. The signed parts are used to protect the integrity of messages. You can specify the signed parts to add for message integrity protection.

The following table shows the required signed parts when the digital signature security constraint (integrity) is defined:

Table 164. Signed parts information. Use the signed parts to secure messages.

Signed parts	Description
keyword	Adds signed parts using keywords. WebSphere Application Server supports the following keywords for signed parts: • BODY • ADDRESSING_HEADERS • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds the required signed parts by using an XPath expression.
header	Adds the header, specified by QName, as a signed part.
timestamp	Adds a WSSTimestamp object as a signed part. If specified, the timestamp information specifies when the message is generated and when it expires.

Different message parts can be specified in the message protection for request on the generator side. WSSSignPart allows for adding a transform algorithm, setting a digest method, setting objects as targets, specifying whether an element, and the signed parts, such as: the SOAP body, the WS-Addressing header, and timestamp information.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignPart API is to use the default behavior (see the example code). The signed parts default behaviors include:

Table 165. Default behavior of signed parts. Several signed part characteristics are configured by default.

Signature decisions	Default behavior
Which SOAP message parts to sign	WebSphere Application Server supports the following SOAP message parts to be signed and used for message protection: • WSSSignature.BODY • WSSSignature.ADDRESSING_HEADERS • WSSSignature.TIMESTAMP
Which digest method to use	Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. WebSphere Application Server supports the following pre-configured digest methods: • WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1 • WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256 • WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512

Table 165. Default behavior of signed parts (continued). Several signed part characteristics are configured by default.

Signature decisions	Default behavior
Which transform algorithms to use	Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms: • WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2 Use this transform method to ensure compliance with the Basic Security Profile (BSP). • WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform • WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

Procedure

1. To enable Web Services Security by using the WSS API (WSSSignPart), first ensure that the application server is installed.
2. Use the WSSSignPart API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signed parts follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken.
   e. Creates WSSSignPart from the WSSFactory instance.
   f. Sets the part to be signed and the digest method or transform method specified by step g or step h if the default is not appropriate.
   g. Sets the digest method if the default is not appropriate.
   h. Sets the transform method if the default is not appropriate.
   i. Adds WSSSignPart to WSSSignature. After any WSSSignPart is set to WSSSignature, the default parts to be signed, which are specified in WSSSignature, are ignored.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signed parts for the generator section of the bindings files. If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses all of methods that are defined in the WSSSignPart API:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance  (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler
      "",
```

```
      "dsig-sender.ks",
      "jks",
      "client".toCharArray(),
      "soaprequester",
      "client".toCharArray(),
      "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token used to the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

// Generate WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();

// Set the part specified by WSSSignPart  (step: f)
   sigPart.setSignPart(WSSSignature.BODY);

// Set the digest method specified by WSSSignPart  (step: g)
   sigPart.setDigestMethod(WSSSignPart.SHA256);

// Set the transform method specified by WSSSignPart  (step: h)
   sigPart.setTransformMethod(WSSSignPart.TRANSFORM_STRT10);

// Add the part specified by WSSSignPart  (step: i)
   sig.addSignPart(sigPart);

// Add the WSSSignature to the WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Use the WSSVerifyPart API or configure policy sets using the administrative console to verify the signed parts on the consumer side.

Configuring request signing methods for the client:

Use the WSSSignature and WSSSignPart APIs to choose the signing methods. The request signing methods include the signature, canonicalization, digest, and transform methods.

Before you begin

First, you must have specified which parts of the message sent by the client must be digitally signed using the WSS APIs or configuring policy sets using the administrative console.

About this task

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 166. Signing methods. Use the signing methods to secure messages.

Name of method	Description
Canonicalization algorithm	Canonicalizes the <SignedInfo> element before the information is digested as part of the signature operation.
Signature algorithm	Calculates the signature value of the canonicalized <SignedInfo> element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Transform method	Transforms the parts to be signed before the information is digested as part of the signature operation.
Digest method	Calculates the digest value of the transformed parts. The algorithm selected for the client request sender configuration must match the algorithms selected in the server request receiver configuration.

You can use the WSS APIs or configure policy sets using the administrative console to configure the signing algorithm methods. If using the WSS APIs, use the WSSSignature and WSSSignPart APIs to specify which message parts to digitally sign when configuring the client for request signing.

The WSSSignature and WSSSignPart APIs complete the following steps to configure the signature and signed part algorithm methods:

Procedure

1. For the generator binding, the WSSSignature API specifies the signature method. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSSignature.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 digital signature method, http://www.w3.org/2000/09/xmldsig#dsa-sha1.
2. For the generator binding, the WSSSignature API specifies the canonicalization method. WebSphere Application Server supports the following pre-configured canonicalization algorithms:
   - WSSSignature.EXC_C14N (the default value): The exclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignature.C14N: The inclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-c14n#
3. For the generator binding, the WSSSignPart API specifies the digest method. WebSphere Application Server supports the following pre-configured digest methods:
   - WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the generator binding, the WSSSignPart API specifies the transform method. WebSphere Application Server supports the following pre-configured transform algorithms:
   - WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:
   - http://www.w3.org/TR/1999/REC-xpath-19991116
   - http://www.w3.org/2002/07/decrypt#XML

Results

Using the WSS APIs, you have specified which algorithm methods are used to digitally sign a message when the client sends a message to a server.

Example

The following example is sample code for specifying the signature information, HMAC_SHA1 as signature method, C14N as a canonicalizaion method, SHA256 as a digest method, and EXC_C14N and TRANSFORM_STRT10 as the transform methods:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();
```

```
//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the digest method
 // DEFAULT: WSSSignPart.SHA1
sigPart.setDigestMethod(WSSSignPart.SHA256);

//add the transform method
// DEFAULT: WSSSignPart.TRANSFORM_EXC_C14N
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_EXC_C14N);
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_STRT10);

 // add the WSSSignPart to the WSSSignature
 sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

After you configure the client to digitally sign the message and to choose the algorithm methods, you must configure the server to verify the digital signature for request signing and to choose the algorithm methods.

Configure policy sets using the administrative console to configure the signature verification information and methods on the server.

Digital signing methods using the WSSSignature API:

You can configure the signing information for the generator binding using the WSS API. To configure the client for request signing, choose the digital signing methods. The algorithm methods include the signing and canonicalization methods.

You must configure generator signing information to protect message integrity by digitally signing SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message.

Methods

Methods that are used for the signing information include the:

Signature method
Sets the signature algorithm method.

Canonicalization method
Sets the canonicalization algorithm method.

Signature algorithms

The signature algorithms specify the algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports the following pre-configured algorithms:

Table 167. Signature algorithms. The algorithms include the signing methods.

Algorithm	Description
WSSSignature.HMAC_SHA1	A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1
WSSSignature.RSA_SHA1 (the default value)	A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1

For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 algorithm, http://www.w3.org/2000/09/xmldsig#dsa-sha1

The signing algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

Table 168. Signature canonicalization algorithms. The algorithms include the canonicalization methods.

Algorithm	Description
WSSSignature.EXC_C14N (the default value)	A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSSignature.C14N	A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#

The canonicalization algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

The following example provides sample WSS API code that specifies the HMAC_SHA1 as a signature method and C14n as a canonicalization method:

```
//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
    X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
```

```
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

Signed parts methods using the WSSSignPart API:

You can configure the signed parts information for the generator binding using the WSS API. The algorithms include the digest and transform methods.

You can protect message integrity by configuring signed parts and key information. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signed parts include the:

Digest method
Sets the digest algorithm method.

Transform algorithm
Sets the transform algorithm method.

Digest algorithms

The digest method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 169. Signed parts digest methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.SHA1 (the default value)	A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1
WSSSignPart.SHA256	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256
WSSSignPart.SHA512	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512

Transform algorithms

The transform method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 170. Signed parts transform methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE	A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature
WSSSignPart.TRANSFORM_STRT10	A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
WSSSignPart.TRANSFORM_EXC_C14N (the default value)	A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n#

Table 170. Signed parts transform methods (continued). The methods are used for the signed parts.

Digest method	Description
WSSSignPart.TRANSFORM_XPATH2_FILTER	A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part.

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code for specifying the signature and signed parts, setting the signing key and adding the STR-Transform transform algorithm as signed parts:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the part specified by WSSSignPart
sigPart.setSignPart(WSSSignature.BODY);

//set the digest method specified by WSSSignPart
sigPart.setDigestMethod(WSSSignPart.SHA256);

//set the transform method specified by WSSSignPart
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);

//set the part specified by WSSSignPart
sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

Attaching the generator token using WSS APIs to protect message authenticity:

When you specify the token generator, the information is used on the generator side to generate the security token.

Before you begin

The token processing and pluggable token architecture in the Web Services Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module

from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Services Security run time.

Restriction: The `com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

Note that the key name (KeyName) element is not supported in the application server because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification.

About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web Services Security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Services Security run time.

Note: WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation
- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the generator side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching consumer security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the generator-side security token, use the appropriate pre-configured token generator interface from the WSS APIs to complete the following token configuration process steps:

Procedure

1. Generate the wssFactory instance.
2. Generate the wssGenerationContext instance.

   The WSSGenerationContext interface stores the components for generating Web Services Security (WS-Security), such as the signing and encryption information, the security token, and the time stamp. When the generate() method is called, all of these components are generated.
3. Create the generator-side components, such as the WSSSignature and the WSSEncryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.
5. Specifiy a token generator class name. The token generator class name specifies the required information to generate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specifiy the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web Services Security header within the SOAP message. Also, the token generator is a plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the WSSGenerationContext interface.

   WebSphere Application Server provides the following default callback handler implementations for the generator side:

   com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback
   This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGUIPromptCallbackHandler
   This class is a callback handler for the Username token with the GUI prompt on the generator side. This instance is used to set the WSSGenerationContext object to generate a Username token.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler
   This class is a callback handler for the Username token on the generator side. This instance is used to set into WSSGenerationContext object to attach a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
   This class is a callback handler that is used to generate the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token on the generator side. This instance is used to generate the WSSSignature and WSSEncryption objects, set the objects into the WSSGenerationContext object to generate the X.509 binary

security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must be provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler
This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the generator side. This instance is used to generate WSSSignature object and WSSEncryption object to generate a LTPA token.

This callback handler is used to validate the LTPA security token inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
This class is a callback handler for the Kerberos v5 token on the generator side. This instance is used to set the WSSGenerationContext object to generate the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSSignature and WSSEncryption objects to use the Kerberos session key or derived key in the SOAP message signature and encryption.

7. If a X.509 token is specified, additional token information is also specified.

Table 171. Information for X.509 token. Use the X.509 token for signing and encryption.

Token Information	Description
storeRef	The reference name of the keystore.
storePath	The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations.
storePassword	The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server.
storeType	The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection: **JKS** Use this option if the keystore uses the Java Keystore (JKS) format. **JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption. **JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only). **PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection. **PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format.
alias	The key alias name. The key alias is used by the key locator to find the key within the keystore file.
keyPassword	The key password that is used for recovering the key. This password is needed to access the key object within the keystore file.
keyName	The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`.
certStores	A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). This step configures a collection certificate store and certificate revocation lists for the generator bindings.

Table 171. Information for X.509 token (continued). Use the X.509 token for signing and encryption.

Token Information	Description
identityAssertion	Specifies whether identity assertion is used. Selects this item if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For an X.509 token generator, the application server sends the original signer certification only.
requestorCertificate	Specifies whether the certificate of the requestor is used.

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain.

d. With keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

Specifies an X.509 certificate token.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2

For an LTPA token, you can use `LTPA` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token generator class name, the following token information can be specified:
   a. Whether to use IdentityAssertion option. This option is selected if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a Username token generator.
   b. Whether to use RunAsSubject identity option. This option is used if an identity assertion is defined and you want to use the Run As identity instead of the initial caller identity for identity assertion in a downstream call. This option is valid only if you have configured the Username token as the token generator.
   c. Whether to use sendRealm.
   d. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the generated token type is a Username token, and it is available only for the request generator binding.
   e. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   f. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

      URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
      Specifies a Username token.
9. If the Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

Token Information	Description	Default Value
name	Kerberos client principal name	
password	Kerberos client password	
realm	Kerberos realm associated with the Kerberos client	Default realm name in Kerberos configuration file. Specify null to use the default value.
targetService	Kerberos service name associated with the target web services.	
targetHost	Kerberos realm name associated with the Kerberos service name.	
tokenValueType	Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification.	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
targetRealm	Kerberos realm name associated with the Kerberos service name.	Default realm name in the Kerberos configuration file
prompt	A boolean value to enable the login prompt.	false

Token Information	Description	Default Value
supportTokenRequireSHA1	A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token.	false SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed.
alwaysAPREQ	A boolean value to indicate that the client should always send the Kerberos AP_REQ token in the request messages.	false The SHA1 key is used instead in the subsequent messages. If set to true, the Kerberos AP_REQ token is always used.
requireDKT	A boolean value to require a derived key for message protection.	false
clabel	The client label for the derived key.	WS-SecureConversation Specify null to use the default value.
slabel	The service label for the derived key.	WS-SecureConversation Specify null to use the default value.
keylen	The length of the derived key.	16 Specify zero to use the default value
noncelen	The length of the nonce.	16 Specify zero to use the default value
encComponent	An instance of WSSEncryption.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.
sigComponent	An instance of WSSSignature.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If Secure Conversation is used for message protection, the following information must be specified:

Information	Description
bootstrapWSSGenerationContext	The bootstrap configuration used to secure the RequestSecurityToken (RST) token.
bootstrapWSSConmingContext	The bootstrap configuration used for consuming a secured RequestSecurityTokenResponse (RSTR).
ENDPOINT_URL	The service end point URL.
EncryptionAlgorithm	This determines the key size.
cLabel	The client label used when creating the derived key.
sLabel	The server label used when creating the derived key.

11. Set the components into the wssGenerationContext object.
12. Invoke the wssGenerationContext.process() method.

Results

Using the Web Services Security API (WSS API) process, you can configured the token generator.

What to do next

Next, you must specify a similar token consumer configuration.

Configuring generator security tokens using the WSS API:

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the generator side, use the Web Services Security APIs (WSS API). The generator security tokens are part of the `com.ibm.websphere.wssecurity.wssapi.token` interface package.

Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security runtime and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must complete the following token tasks:

- Configure the generator tokens.
- Configure the consumer tokens.

About this task

The JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token on the generator side.

On the generator side, the token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Services Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Services Security runtime to the LoginModule. After the token is authenticated, a security token object is created, and the token is passed it to the Web Services Security runtime.

When using the WSS API for generator token creation, certain default behaviors occur. The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide default values for the token type, the token value, and the JAAS confirmation name. The default token behaviors include:

Table 172. Token decisions and default behaviors. Several token characteristics are configured by default.

Generator token decisions	Default behavior
Which token type to use	The token type specifies which type of token to use for message integrity, message confidentiality, or message authenticity. WebSphere Application Server provides the following pre-configured generator token types for message integrity and message confidentiality: • Derived key token • X509 tokens You can also create custom token types, as needed. WebSphere Application Server also provides the following pre-configured generator token types for the message authenticity: • Username token • LTPA tokens • X509 tokens You can also create custom token types, as needed.
What JAAS login configuration name to specify	The JAAS login configuration name specifies which JAAS login configuration name to use.
Which configuration type to use	The JAAS login module specifies the configuration type. Only the pre-configured generator configuration types can be used for generator token types.

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, the XML format, and the cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following tokens types are subclasses of the generic security token class:

Table 173. Subclasses of the SecurityToken. Use the subclasses to represent the security token.

Token type	JAAS login configuration name
Username token	system.wss.generate.unt
Security context token	system.wss.generate.sct
Derived key token	system.wss.generate.dkt

The following tokens types are subclasses of the binary security token class:

Table 174. Subclasses of the BinarySecurityToken. Use the subclasses to represent the binary security token.

Token type	JAAS login configuration name
LTPA token	system.wss.generate.ltpa
LTPA propagation token	system.wss.generate.ltpaProp
X.509 token	system.wss.generate.x509
X.509 PKI Path token	system.wss.generate.pkiPath
X.509 PKCS7 token	system.wss.generate.pkcs7

Note:

- For each JAAS login token generator configuration name, there is a respective token consumer configuration name. For example, for the Username token, the respective token consumer configuration name is system.wss.consume.unt.

- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

Procedure

1. To configure the securityToken package, `com.ibm.websphere.wssecurity.wssapi.token`, first ensure that the application server is installed.
2. Use the Web Services Security token generator process to configure the tokens. For each token type, the process is similar to the following process that demonstrates the UsernameToken token generator process:
   a. Use WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Create the WSSGenerationContext instance from the WSSFactory instance.
   c. Create a JAAS CallbackHandler. The authentication data, such as the user name and password are specified as part of the CallbackHandler. For example, the following code specifies Chris as the user name and sirhC as the password: UNTGenerationCallbackHandler("Chris", "sirhC");
   d. Call any JAAS CallbackHandler parameters and review the token class information for which parameters are required or optional. For example, for the UsernameToken, the following parameters can be configured also:

      Nonce
      Indicates whether a nonce is included in the user name token for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The nonce value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

      Created timestamp
      Indicates whether to insert a time stamp into the UsernameToken. The timestamp value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

   e. Create the SecurityToken from WSSFactory.

      By default, the UsernameToken API specifies the ValueType as: "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"

      By default, the UsernameToken API provides the QName of this class and specifies the NamespaceURI as http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd and also specifies the LocalPart as UsernameToken.
   f. Optional: Specify the JAAS login module configuration name. On the generator side, the configuration type is always `generate` (for example, `system.wss.generate.unt`).
   g. Add the SecurityToken to the WSSGenerationContext.
   h. Call WSSGenerationContext.process() and generate the WS-Security header.

Results

If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext process() is called, and the security token for the generator binding is attached.

Example

The following example code shows how to use WSS APIs to create a Username security token, attach the Username token to the SOAP message, and configure the Username token in the generator binding.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
```

```
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

   // generate WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

   // generate callback handler
   UNTGenerateCallbackHandler untCallbackHandler =
   new UNTGenerateCallbackHandler("Chris", "sirhC");

   // generate the username token
   SecurityToken unt = factory.newSecurityToken(UsernameToken.class, untCallbackHandler);

   // add the SecurityToken to the WSSGenerationContext
   gencont.add(unt);

   // generate the WS-Security header
   gencont.process(reqContext);
```

The following example code shows how to modify the preceding Username token sample to create an LTPAv2 token from the runAs identity on the current thread. The two lines of code that instantiate the callback handler and create the security token are replaced with the following two lines of code:

```
// generate callback handler
LTPAGenerateCallbackHandler ltpaCallbackHandler = new LTPAGenerateCallbackHandler(null, null);

// generate the LTPAv2 token
SecurityToken ltpa = wssfactory.newSecurityToken(LTPAv2Token.class, ltpaCallbackHandler);
```

The instantiation of the LTPAGenerateCallbackHandler object with (null, null) indicates that the LTPA token should be generated from the current runAs identity. If the callback handler is instantiated with basicAuth information, ("userName", "password"), a new LTPA token is created using the specified basicAuth information.

The following example shows how to use secure conversation with the WSS APIs to configure the generator tokens, as well as the consumer tokens. In this example, the SecurityContextToken token is created using the WS-SecureConversation draft namespace: `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`. To use the WS-SecureConversation version 1.3 namespace, `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`, specify SecurityContextToken13.class instead of SecurityContextToken.class.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory wssFactory = WSSFactory.getInstance();

   WSSGenerationContext bootstrapGenCon = wssFactory.newWSSGenerationContext();

   // Create a Timestamp
   ...
   // Add Timestamp
   ...

   // Sign the SOAP Body, WS-Addressing headers, and Timestamp
   X509GenerateCallbackHandler btspReqSigCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqSigToken = wssFactory.newSecurityToken(X509Token.class,
                                                               btspReqSigCbHandler);
   WSSSignature bootstrapReqSig = wssFactory.newWSSSignature(btspReqSigToken);
   bootstrapReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);

   // Add Sign Parts
   ...
   bootstrapGenCon.add(bootstrapReqSig);

   // Encrypt the SOAP Body and the Signature
   X509GenerateCallbackHandler btspReqEncCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqEncToken = wssFactory.newSecurityToken(X509Token.class,
```

```
                                                        btspReqEncCbHandler);
WSSEncryption bootstrapReqEnc = wssFactory.newWSSEncryption(btspReqEncToken);
bootstrapReqEnc.setEncryptionMethod(WSSEncryption.AES128);
bootstrapReqEnc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);

// Add Encryption parts
...
bootstrapGenCon.add(bootstrapReqEnc);
WSSConsumingContext bootstrapConCon = wssFactory.newWSSConsumingContext();
X509ConsumeCallbackHandler btspRspVfyCbHandler = new X509ConsumeCallbackHandler(....);
WSSVerification bootstrapRspVfy = wssFactory.newWSSVerification(X509Token.class,
                                                          btspRspVfyCbHandler);
bootstrapRspVfy.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Add Verify parts
...
bootstrapConCon.add(bootstrapRspVfy);
X509ConsumeCallbackHandler btspRspDecCbHandler = new X509ConsumeCallbackHandler(...);
WSSDecryption bootstrapRspDec = wssFactory.newWSSDecryption(X509Token.class,
                                                        btspRspDecCbHandler);
bootstrapRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
bootstrapRspDec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA15);

// Add Decryption parts
...
bootstrapConCon.add(bootstrapRspDec);
SCTGenerateCallbackHandler sctgch = new SCTGenerateCallbackHandler(bootstrapGenCon,
                                                                   bootstrapConCon,
                                                                   ENDPOINT_URL,
                                                                   WSSEncryption.AES128);
SecurityToken[] scts = wssFactory.newSecurityTokens(new Class[]{SecurityContextToken.class},
                                                    sctgch);
SecurityContextToken sct = (SecurityContextToken)scts[0];

// Use the SCT to generate DKTs for Secure Conversation
// Signature algorithm and client and service labels
DerivedKeyToken dktSig = sct.getDerivedKeyToken(WSSSignature.HMAC_SHA1,
                                                "WS-SecureConversation",
                                                "WS-SecureConversation");

// Encryption algorithm and client and service labels
DerivedKeyToken dktEnc = sct.getDerivedKeyToken(WSSEncryption.AES128,
                                                "WS-SecureConversation",
                                                "WS-SecureConversation");

// Create the application generation context for the request message
WSSGenerationContext applicationGenCon = wssFactory.newWSSGenerationContext();

// Create and add Timestamp
...

// Add the derived key token and Sign the SOAP Body and WS-Addressing headers
WSSSignature appReqSig = wssFactory.newWSSSignature(dktSig);
appReqSig.setSignatureMethod(WSSSignature.HMAC_SHA1);
appReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
...
applicationGenCon.add(appReqSig);

// Add the derived key token and Encrypt the SOAP Body and the Signature
WSSEncryption appReqEnc = wssFactory.newWSSEncryption(dktEnc);
appReqEnc.setEncryptionMethod(WSSEncryption.AES128);
appReqEnc.setTokenReference(SecurityToken.REF_STR);
appReqEnc.encryptKey(false);
...
applicationGenCon.add(appReqEnc);

// Create the application consuming context for the response message
WSSConsumingContext applicationConCon = wssFactory.newWSSConsumingContext();

//client and service labels and decryption algorithm
SCTConsumeCallbackHandler sctCbHandler = new SCTConsumeCallbackHandler("WS-SecureConversation",
                                                                       "WS-SecureConversation",
                                                                       WSSDecryption.AES128);

// Derive the token from SCT and use it to Decrypt the SOAP Body and the Signature
WSSDecryption appRspDec = wssFactory.newWSSDecryption(SecurityContextToken.class,
                                                      sctCbHandler);
appRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
appRspDec.encryptKey(false);
...
applicationConCon.add(appRspDec);

// Derive the token from SCT and use it to Verify the
// signature on the SOAP Body, WS-Addressing headers, and Timestamp
WSSVerification appRspVfy = wssFactory.newWSSVerification(SecurityContextToken.class,
                                                    sctCbHandler);
...
```

```
applicationConCon.add(appRspVfy);
...
applicationGenCon.process(reqContext);
applicationConCon.process(reqContext);
```

What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar consumer tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by signing the SOAP message or by encrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

Securing messages at the request generator using WSS APIs:

You can secure SOAP messages by configuring signing information, encryption, and generator tokens to protect message integrity, confidentiality, and authenticity, respectively. This request (client-side) generator configuration defines the Web Services Security requirements for the outgoing SOAP message request.

Before you begin

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. Therefore, in addition to securing messages at the request generator level, you must also secure messages at the response consumer level.

About this task

The request (client-side) generator configuration requirements involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches security tokens.

To secure web service applications, you must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

You can use the following interfaces to configure Web Services Security and to define policy types to secure the SOAP messages:

- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:

Procedure

- Configure generator signing to protect message integrity.
- Configure encryption to protect message confidentiality.
- Attach generator tokens to protect message authenticity.
- Propagate self-issued SAML bearer tokens using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with message protection using WSS APIs.
- Propagate self-issued SAML sender-vouches tokens with transport protection using WSS APIs.
- “Sending self-issued SAML holder-of-key tokens with symmetric key using WSS APIs” on page 1598.
- “Sending self-issued SAML holder-of-key tokens with asymmetric key using WSS APIs” on page 1600.

Results

After completing these procedures, you have secured messages at the request generator level.

What to do next

Next, if not already configured, secure messages with signature verification, decryption, and consumer tokens at the response consumer (client-side) level.

Configuring encryption to protect message confidentiality using the WSS APIs:

You can configure encryption information for the client-side request generator (sender) bindings. Encryption information is used to specify how the generators (senders) encrypt outgoing SOAP messages. To configure encryption, specify which message parts to encrypt and specify which algorithm methods and security tokens are to be used for encryption.

Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. Prior to configuring encryption, familiarize yourself with XML encryption.

About this task

For encryption, you must specify the following:

- Which parts of the message are to be encrypted.
- Which encryption algorithms to specify.

To configure encryption and encrypted parts on the client side, use the WSSEncryption and WSSEncryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses encryption information for the default generator to encrypt parts of the SOAP message. The WSSEncryption API configures the following required parts as encrypted parts.

Table 175. Required encrypted parts. Use encrypted parts to increase the confidentiality of SOAP messages.

Encryption parts	Description
Keywords	Keywords are used to add the encrypted parts to the SOAP message.
XPath expression	An XPath expression is used to add the encrypted parts to the SOAP message.
WSSEncryptPart object	This object adds the encrypted parts to the SOAP message.
WSSSignature object	This object adds the signature component as an encrypted part.
Header	This part adds the header in the SOAP header, specified by QName, as an encryption part.
Security token object	This object adds the security token as an encryption part.

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the message that are to be encrypted. WebSphere Application Server supports the use of the following keywords:

Table 176. Supported encryption keywords. Use keywords to specify encrypted parts.

Keyword	References
BODY_CONTENT	The keyword for the contents of the SOAP message body as an encryption target.
SIGNATURE	The keyword for the signature component as an encryption target.

If configuring using the WSS APIs, the WSSEncryption and WSSEncryptPart APIs complete these high-level steps:

Procedure

1. Use the WSSEncryption API to configure encryption. The WSSEncryption API performs these tasks by default:
   a. Generates the callback handler.
   b. Generates the generator security token object.
   c. Adds the security token reference type.
   d. Adds the signature component.
   e. Adds the WSSEncryptPart object.
   f. Adds the parts to be encrypted. Adds the default parts as targets of encryption by using keywords and XPath expressions.
   g. Adds the header in the SOAP message, specified by QName.
   h. Sets the default data encryption method.
   i. Specifies whether the key is to be encrypted using a Boolean value.
   j. Sets the default key encryption method.
   k. Selects a part reference.
   l. Sets the MTOM optimization Boolean value.
2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the key (false).
   - Change the security token type from default of X.509 token.
   - Change the security token reference type from the default value of SecurityToken.REF_STR.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.
   - Turn MTOM optimization on (true).

Results

The encryption information is configured for the generator binding.

Example

The following is an example of the WSSEncryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSGenerationContext gencont = factory.newWSSGenerationContext();

X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);
WSSEncryption enc = factory.newWSSEncryption(token);

gencont.add(enc);
```

What to do next

You must configure similar decryption information for the client-side response consumer (receiver) bindings, if you have not already configured the information.

Next, review the WSSEncryption API process.

Encrypting the SOAP message using the WSSEncryption API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for request encryption on the generator side, use the WSSEncryption API to encrypt the SOAP message. The WSSEncryption API specifies which request SOAP message parts to encrypt when configuring the client.

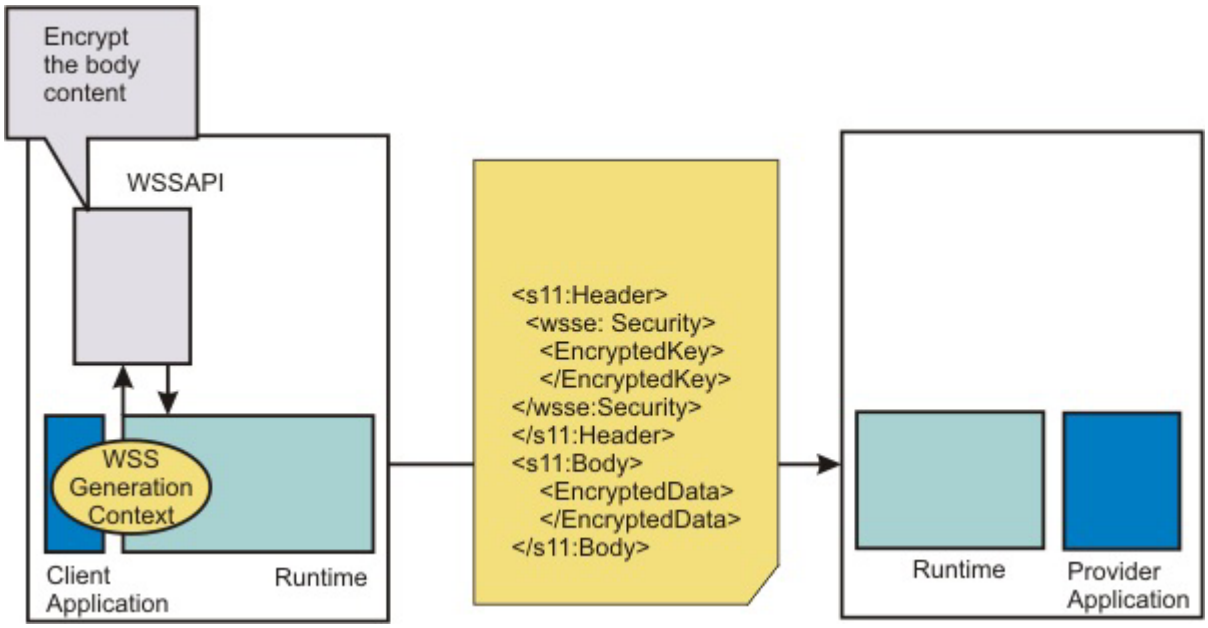
Before you begin

You can use the WSS API or use policy sets on the administrative console to enable encryption and add generator security tokens in the SOAP message. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts, as needed, using the WSSEncryptPart API.

About this task

The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The client generator configuration must match the configuration for the provider consumer.



Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and which message parts to attach encrypted Nonce and timestamp elements to.

The following encryption parts can be configured:

Table 177. Encryption parts. Use the encryption parts to enable encryption in messages.

Encryption parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encryption parts using keywords. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encryption part using an XPath expression.
signature	Adds the WSSignature component as a target of the encrypted part.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encryption, certain default behaviors occur. The simplest way to use the WSSEncryption API is to use the default behavior (see the example code).

WSSEncryption provides defaults for the key encryption algorithm, the data encryption algorithm, the security token reference method, and the encryption parts such as the SOAP body content and the signature. The encryption default behaviors include:

Table 178. Encryption decisions. Use encryption default behavior to secure the message body content and signature.

Encryption decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Sets the encryption parts that you can add using keywords. The default encryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE

Table 178. Encryption decisions (continued). Use encryption default behavior to secure the message body content and signature.

Encryption decisions	Default behavior
Which data encryption method to choose (algorithm)	Sets the data encryption method. Both data and key encryption methods can be specified. The default data encryption algorithm method is AES 128. WebSphere Application Server supports these data encryption methods: • WSSEncryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc • WSSEncryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc • WSSEncryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc • WSSEncryption.TRIPLE_DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc
Whether to encrypt the key (isEncrypt)	Specifies whether to encrypt the key. The values are `true` or `false`. The default value is to encrypt the key (`true`).
Which key encryption method to choose (algorithm)	Sets the key encryption method. Both data and key encryption methods can be specified. The default key encryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods: • WSSEncryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128 • WSSEncryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192 • WSSEncryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256 • WSSEncryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p • WSSEncryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5 • WSSEncryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes
Which security token to specify (securityToken)	Sets the SecurityToken. The default security token type is the X509Token. WebSphere Application Server provides the following pre-configured consumer token types: • Derived key token • X.509 tokens
Which token reference to use (refType)	Sets the type of the security token reference. The default token reference is SecurityToken.REF_KEYID. WebSphere Application Server supports the following token reference types: • SecurityToken.REF_KEYID • SecurityToken.REF_STR • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT
Whether to use MTOM (mtomOptimize)	Sets Message Transmission Optimization Mechanism (MTOM) optimization for the encrypted part.

Procedure

1. To encrypt the SOAP message using the WSSEncryption API, first ensure that the application server is installed.
2. The WSS API process for encryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory used for encryption.
   d. Creates WSSEncryption from the WSSFactory instance using the SecurityToken. The default behavior of WSSEncryption is to encrypt the body content and the signature.
   e. Adds a new part to be encrypted in WSSEncryption if the existing part is not appropriate. After addEncryptPart(), addEncryptHeader(), or addEncryptPartByXPath() is called, the default part is cleared.
   f. Calls the encryptKey(false) if the key is not to be encrypted.
   g. Sets the data encryption method if the default method is not appropriate.
   h. Sets the key encryption method if the default method is not appropriate.
   i. Sets the token reference if the default token reference is not appropriate.
   j. Adds WSSEncryption to WSSConsumingContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample code using methods that are defined in WSSEncryption:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance (step: b)
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
X509GenerateCallbackHandler callbackHandler = new
         X509GenerateCallbackHandler(
     "",
     "enc-sender.jceks",
     "jceks",
     "storepass".toCharArray(),
     "bob",
     null,
     "CN=Bob, O=IBM, C=US",
     null);

// Generate the security token used for encryption (step: c)
SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);

// Generate WSSEncryption instance (step: d)
WSSEncryption enc = factory.newWSSEncryption(token);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
   enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
   enc.addEncryptHeader(new QName("http://www.w3.org/2005/08/addressing",
                                  "MessageID"));

// Set the part specified by WSSSignature (step: e)
   SecurityToken sigToken = getSecurityToken();
      WSSSignature sig = factory.newWSSSignature(sigToken);
   enc.addEncryptPart(sig);

// Set the part specified by SecurityToken (step: e)
   UNTGenerateCallbackHandler untCallbackHandler =
          new UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
                                                   untCallbackHandler);
   enc.addEncryptPart(unt, false);

// sSt the part specified by XPath expression (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
   enc.addEncryptPartByXPath(sb.toString());

// Set whether the key is encrypted (step: f)
// DEFAULT: true
   enc.encryptKey(true);

// Set the data encryption method (step: g)
// DEFAULT: WSSEncryption.AES128
   enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method (step: h)
// DEFAULT: WSSEncryption.KW_RSA_OAEP
   enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Set the token reference (step: i)
// DEFAULT: SecurityToken.REF_KEYID
```

```
  enc.setTokenReference(SecurityToken.REF_STR);

// Add the WSSEncryption to the WSSGenerationContext (step: j)
  gencont.add(enc);

// Process the WS-Security header (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler does not need the key password because the public key is used for encryption. You do not need a password to obtain the public key from the Java keystore.

What to do next

If you have not previously specified which encryption methods to choose, use the WSS API or configure the policy sets using the administrative console to choose the data and key encryption algorithm methods.

Choosing encryption methods for generator bindings:

To configure the client for request encryption for the generator binding, you must specify which encryption methods to use when the client encrypts the SOAP messages.

Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To specify which algorithm methods are to be used when the client encrypts the SOAP messages, complete the following tasks:

- Use the WSSEncryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSEncryptPart API to configure a transform algorithm method, if needed. The default is no transform algorithm.

About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

Data encryption method algorithm

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key encryption method algorithm

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the method to encrypt the key that is used to encrypt data. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property.

The property name is: com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Table 179. Encryption usage types. The encryption usage types describe encryptions methods.

Usage types	Description
Data encryption	Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks.
Key encryption	Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

Data encryption

WebSphere Application Server supports the following pre-configured data encryption algorithms:

Table 180. Data encryption algorithms. These pre-configuring encryption algorithms are supported by WebSphere Application Server.

Data encryption name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, 3DES: http://www.w3.org.2001/04/xmlenc#tripledes-cbc

Key encryption

WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 181. Key encryption algorithms. These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

Table 181. Key encryption algorithms (continued). These pre-configured encryption algorithms are supported by WebSphere Application Server.

Key encryption name	Algorithm URI
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	http://www.w3.org/2001/04/xmlenc#kw-tripledes

To configure the encryption and encrypted part algorithm methods, use the WSSEncryption API, or configure policy sets using the administrative console.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints that use policy sets.

The WSS API process completes the following high-level steps to specify which encryption methods to use when configuring the client for request encryption:

Procedure

1. Using the WSSEncryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The client generator configuration must match the configuration for the provider consumer.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data encryption algorithms:

   - AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - TRIPLEDES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
// Default data encryption algorithm: AES128
WSSEncryption enc = factory.newWSSEncryption(x509t);
   enc.setEncryptionMethod(EncryptionMethod.TRIPLEDES_CBC);
gencont.add(enc);
```

3. Using the WSSEncryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. If the encryption key, which is the key that is used for encrypting the message parts, is not encrypted, then the decryption API selects `false` to match the encryption key.

   The client generator configuration must match the configuration for the provider consumer.

The default key encryption algorithm value is key wrap RSA OAP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

- KW AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
- KW AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

  To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

  Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).KW AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256

  To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
- KW RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

  The KW RSA OAEP algorithm is the default key algorithm method.

  When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
- KW RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
- KW TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

Note: For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
- algorithmName
- keyLength

Results

If there is an error condition, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample WSS API code using WSSEncryption.setEncryptionMethod() and WSSEncryption.setKeyEncryptionMethod().

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
                                   "",
                                   "enc-sender.jceks",
                                   "jceks",
                                   "storepass".toCharArray(),
                                   "bob",
                                   null,
                                   "CN=Bob, O=IBM, C=US",
                                   null);

// Generate the security token used for encryption
   SecurityToken token = factory.newSecurityToken(X509Token.class , callbackHandler);
```

```
// Generate WSSEncryption instance
   WSSEncryption enc = factory.newWSSEncryption(token);

// Set the data encryption method
// DEFAULT: WSSEncryption.AES128
      enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// Set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
      enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// Add the WSSEncryption to the WSSGenerationContext
   gencont.add(enc);

// Generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

Next, if you want to add a transform algorithm, review the WSSEncryptPart API process task.

Encryption methods:

For request generator binding settings, the encryption methods include specifying the data and key encryption algorithms to use to encrypt the SOAP message. The WSS API for encryption (WSSEncryption) specifies the algorithm name and the matching algorithm uniform resource identifier (URI) for the data and key encryption methods. If the data and key encryption algorithms are specified, only elements that are encrypted with those algorithms are accepted.

Data encryption algorithms

The data encryption algorithm is used to encrypt parts of the SOAP message, including the body and the signature. Data encryption algorithms specify the algorithm uniform resource identifier (URI) for each type of data encryption algorithms.

The following pre-configured data encryption algorithms are supported:

Table 182. Data encryption algorithms. The algorithms are used to encrypt SOAP messages.

Data encryption algorithm name	Algorithm URI
WSSEncryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSEncryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSEncryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSEncryption.TRIPLE_DES	A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For the AES256-cbc and the AES192-CBC algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm configured for encryption for the generator side must match the data encryption algorithm that is configured for decryption for the consumer side.

Key encryption algorithms

This algorithm is used to encrypt and decrypt keys. This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints using the policy sets.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The following pre-configured key encryption algorithms are supported:

Table 183. Supported pre-configured key encryption algorithms. The algorithms are used to encrypt and decrypt keys.

WSS API	URI
WSSEncryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSEncryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSEncryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSEncryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSEncryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSEncryption.KW_TRIPLE_DES	A URI of key encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

For Secure Conversation, additional key-related information must be specified, such as:

- algorithmName
- keyLength

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW-AES256 and the KW-AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator must match the key decryption algorithm that is configured for the consumer.

This example provides sample code for encryption to use the Triple DES for the data encryption method and to use RSA1.5 for the key encryption method:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "enc-sender.jceks",
  "jceks",
  "storepass".toCharArray(),
  "bob",
  null,
  "CN=Bob, O=IBM, C=US",
  null);

// generate the security token used to the encryption
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

// generate WSSEncryption instance to encrypt the SOAP body content
WSSEncryption enc = factory.newWSSEncryption(token);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);

// set the data encryption method
// DEFAULT: WSSEncryption.AES128
enc.setEncryptionMethod(WSSEncryption.TRIPLE_DES);

// set the key encryption method
// DEFAULT: WSSEncryption.KW_RSA_OAEP
enc.setEncryptionMethod(WSSEncryption.KW_RSA15);

// add the WSSEncryption to the WSSGenerationContext
gencont.add(enc);

// generate the WS-Security header
gencont.process(msgcontext);
```

Adding encrypted parts using the WSSEncryptPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure encrypted parts for the request generator (client side) bindings, use the WSSEncryptPart API to define and add to the listing of elements in the encrypted part. WSSEncryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.encryption package.

Before you begin

You can use the WSS APIs or configure policy sets using the administrative console to enable the encrypted parts. To secure SOAP messages, use the WSS APIs to complete the following encryption tasks, as needed:

- Configure encryption and choose the encryption methods using the WSSEncryption API.
- Configure the encrypted parts using the WSSEncryptpart API, as needed.

About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted, and

which message parts to attach encrypted elements to. The encryption information on the generator side is used for encrypting an outgoing SOAP message. The request generator is configured for the client.

The WSSEncryptPart API specifies information related to encrypted parts and sets the encrypted parts that have been added for message confidentiality protection. Use the WSSEncryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSEncryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The encrypted parts and related information displayed in the following table are used to protect the confidentiality of messages.

Table 184. Encrypted parts. Use encrypted parts to secure messages.

Encrypted parts	Description
part	Adds the WSSEncryptPart object as a target of the encryption part.
keyword	Adds the encrypted parts using keywords. The default encryption parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • BODY_CONTENT • SIGNATURE
xpath	Adds the encrypted part by using an XPath expression.
signature	Adds the WSSSignature component as a target of the encrypted part. WSSSignature is applicable only if the SOAP message contains a signature element.
header	Adds the SOAP header, specified by QName, as a target of the encrypted part.
securityToken	Adds the SecurityToken object as a target of the encrypted part.

For encrypted parts, certain default behaviors occur. The simplest way to use the WSSEncryptPart API is to use the default behavior. The WSSEncryptPart API provides defaults for specifying the transform algorithm, setting objects as targets, specifying the encrypted parts, such as: the SOAP body content and the signature.

The encryption default behaviors include:

Table 185. Encrypted part decisions. Several encrypted message parts are set by default.

Encrypted part decisions	Default behavior
Which SOAP message parts to encrypt using keywords	Specifies which keywords to use for the encrypted parts. WebSphere Application Server sets the following SOAP message parts by default for encryption: • WSSEncryption.BODY_CONTENT • WSSEncryption.SIGNATURE
Which transform method to add	WebSphere Application Server does not specify any transform method by default. Specify a transform method only if using SOAP with Attachments.

Procedure

1. To encrypt the SOAP message parts using the WSSEncryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSEncryptPart follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the encryption.
   d. Creates WSSEncryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSEncryptPart from WSSFactory.
   f. Adds the parts to be encrypted and to be applied with the transform in WSSEncryptPart. WebSphere Application Server sets these encrypted parts by default for WSSEncryptPart: the BODY_CONTENT and SIGNATURE. After you add other encrypted parts, the default values are no

longer valid. For example, if you call addEncryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to encrypt the security token, the signature, and the body content, you must call addEncryptPart(securityToken, false), addEncryptPart(WSSEncryption.SIGNATURE), and addEncryptPart(WSSEncrypyion.BODY_CONTENT).

g. Sets the transform method.
h. Adds WSSEncryptPart to WSSEncryption.
i. Adds WSSEncryption to WSSGenerationContext.
j. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

If there is an error condition during encryption of the message parts, a WSSException is provided. If successful, the API calls the WSSGenerationContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

What to do next

After enabling encrypted parts for the request generator (client side) binding, you must specify the same parts to be decrypted for the response consumer (client side) bindings. Next, to configure decryption and decrypted parts, use the WSS APIs or configure policy sets using the administrative console.

Configuring generator signing information to protect message integrity using the WSS APIs:

You can configure the signing information to protect message integrity for the request (client side) generator binding. Signing information includes the signature and the signed parts. To keep the integrity of the message, digital signatures are typically applied.

Before you begin

In addition to using a digital signature and configuring the signing information, the following tasks should also be performed:

- Verify the signing information.
- Incorporate encryption.
- Attach security tokens.

About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by digitally signing the body, time stamp, and WS-Addressing headers using the signature algorithm methods. The WSS APIs specify which algorithm is to be used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports several pre-configured request signing algorithm methods.

You can use the following interfaces to configure Web Services Security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for the signing information.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client).

Perform the following signing tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the generator binding.

Procedure

- Configure the signing information using the WSSSignature API. Configure the signing information for the generator binding using the WSSSignature API. Signing information is used to sign parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both signing and encryption can be applied to the same message parts, such as the SOAP body.
- Add or change signed parts using the WSSSignPart API.
- Configure the client for request signing methods using the WSSSignature or WSSSignPart APIs. To configure the client for request signing, choose the signing methods. The request signing methods include the signature, the canonicalization, the digest, and the transform methods. Use the WSSSignature API to configure the signature and canonicalization methods. Use the WSSSignPart API to configure the digest and transform methods.

Results

The WSS APIs also specify the security token for the generator (client) binding and set the type of token reference to protect message authenticity. By completing the steps in these tasks, you have configured generator signing to protect the integrity of the SOAP message.

What to do next

Next, verify the consumer signing information by using the WSS APIs or by configuring policy sets using the administrative console.

Configuring signing information using the WSS APIs:

You can configure the signing information for the client-side request generator (sender) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token. To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method. Prior to completing these steps, familiarize yourself with XML digital signature for signing and verifying digital signatures for digital content.

About this task

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. WebSphere Application Server uses the signing information for the default generator to sign parts of the message, such as the body, time stamp, and Username token.

For the signing information, you must specify the following:

- Which parts of the message are to be signed.
- The key information that is referenced by the key information for the signing keys.
- The signing algorithms.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSSignature API configures the following parts as signature parts:

Table 186. Pre-configured signature parts. Use the signing information to validate parts of a message.

Part	Description
Security token object	This object authenticates the client. If this option is specified, then the message is signed. You can digitally sign the message using a security token if a login configuration authentication method is selected.
WSSTimestamp object	This object adds a time stamp to a message. The time stamp determines if the message is valid based on the time that the message is sent and then received.
WSSSignature Part object	This object adds the signature parts to a message.
SOAP header and the QName as a target	This signature part adds the header, specified by QName, as a verification part.

The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be signed. WebSphere Application Server supports the use of the following keywords:

Table 187. Supported signature keywords. Key information is used to specify which parts of a message are signed.

Keyword	References
ADDRESSING_HEADERS	The Web Services Addressing (WS-Addressing) headers.
BODY	The SOAP message body. The body is the user data portion of the message.
TIMESTAMP	The creation and expiration timestamp information.

The Web Services Security API (WSS API) are used to configure the signing information for the request generator (client side) section of the bindings file. To configure the signing information on the client side, use the WSS APIs or configure policy sets for signing using the administrative console.

If configuring using the WSS APIs, the WSSSignature and WSSSignPart APIs complete the following steps to specify which message parts to digitally sign when configuring the client for request generator signing:

Procedure

1. The WSSSignature API adds the required parts of the SOAP message to digitally sign. Either a keyword or an XPath expression can be used to specify the required encryption parts.
2. The WSSSignature API sets the signature method algorithm. The default signature method is RSA_SHA1. WebSphere Application Server supports the following pre-configured algorithms:
   - RSA SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - HMAC SHA1http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).

   Any ds:SignatureMethod/@Algorithm element in a signature is based on a symmetric key and must have a value of RSA-SHA1 or HMAC-SHA1.

   The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.
3. The WSSSignature API sets the canonicalization method. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured algorithms:
   - The URI of the exclusive canonicalization algorithm, EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#.
   - The URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#.

   The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.
4. The WSSSignature API adds a security token. The API adds information about the security token that is to be used for the signature, such as:
   - The class for security token.

- The callback handler
- The name of the JAAS login configuration.

5. The WSSSignature API sets the type of security token and sets the type of token reference. WebSphere Application Server supports the following pre-configured token references:
   - SecurityToken.REF_STR

     Represents the security token reference as a token reference type.
   - SecurityToken.REF_KEYID

     Represents the key identifier reference as a token reference type.
   - SecurityToken.REF_EMBEDDED

     Represents the embedded reference as a token reference type.
   - SecurityToken.REF_THUMBPRINT

     Represents the thumbprint reference as a token reference type.
6. If SecurityToken.REF_KEYID is set as the type of token reference, the WSSSignature API sets the key information signature type and configures the key information that is referenced by the key information references. WebSphere Application Server supports the following:
   - Specifying that the KeyInfo element is not signed.
   - Specifying that the entire <KeyInfo> element is signed.
   - Specifying that the child elements <Keyinfochildelements> of the <KeyInfo> element are signed.

   If you do not specify one of the previous signature types, WebSphere Application Server specifies that the entire <KeyInfo> element is signed, by default.

   If you select Keyinfo or Keyinfochildelements and you select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm in a subsequent step, WebSphere Application Server also signs the referenced token.

   The key information signature type for the generator must match the signature type for the consumer.
7. The WSSSignature API specifies whether to require signature confirmation. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

   The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.
8. The WSSSignPart API specifies the part reference. The part reference specifies which parts of the message to digitally sign.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature verification: <DigestTransform> and <Transform>.
9. The WSSSignPart API specifies the digest method algorithm. The digest method algorithm specified within the <DigestMethod> element is used in the <SigningInfo> element.

   WebSphere Application Server supports the following pre-configured digest algorithms:
   - http://www.w3.org/2000/09/xmldsig#sha1
   - http://www.w3.org/2001/04/xmlenc#sha256
   - http://www.w3.org/2001/04/xmlenc#sha512
10. The WSSSignPart API specifies the transform algorithm. The transform algorithm is that is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms:
    - http://www.w3.org/2001/10/xml-exc-c14n#
    - http://www.w3.org/TR/1999/REC-xpath-19991116

Do not use this transform algorithm if you want to be compliant with the Basic Security Profile (BSP). Instead use http://www.w3.org/2002/06/xmldsig-filter2 to ensure compliance.

- http://www.w3.org/2002/06/xmldsig-filter2
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
- http://www.w3.org/2002/07/decrypt#XML
- http://www.w3.org/2000/09/xmldsig#enveloped-signature

The transform algorithm that you select for the generator must match the transform algorithm that you select for the consumer.

Important: If both of the following conditions are true, WebSphere Application Server signs the referenced token:

- You previously selected the Keyinfo or the Keyinfochildelements option
- You select http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform as the transform algorithm.

11. If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. Configure policy sets using the administrative console to configure the same actor strings for the web service on the server, which processes the request and sends the response back.

    The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

Results

After the WSSSignature and WSSSignPart APIs complete these steps, the signing information is configured for the generator sections of the bindings files.

Example

The following example shows WSS API sample code to configure the signature, to generate the callback handler, and to specify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
// Instantiate a generation context
WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler and specify the X.509 token
   X509GenerateCallbackHandler callbackhandler = generateCallbackHandler();
   SecurityToken token = factory.newSecurityToken(X509Token.class,
                                                  callbackHandler);

// Set the signature information
WSSSignature sig = factory.newWSSSignature(token);
// Add the header using QName
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "To"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));
   sig.addSignHeader(new QName("http://www.w3.org/2005/08/addressing", "Action"));
// Apply the signature
   gencont.add(sig);

// Secure the message
gencont.process(msgctx);
```

What to do next

You must configure similar signature information for the client-side request consumer (receiver) bindings by completing the following verification tasks:

- Verify the signature
- Choose the signature algorithm methods.
- Change or add signed parts, as needed.

If signature verification is already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

Configuring signing information using the WSSSignature API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the signing information for the generator binding sections for the client-side request, use the WSSSignature API. The WSSSignature API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to enable the signing information. To secure SOAP messages, you must complete the following signing tasks:

- Configure the signing information.
- Choose the signing methods.
- Add or change signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-SHA1.

XML signature defines many methods for describing key information and enables the definition of a new method. XML canonicalization (C14N) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The signing information specifies the integrity constraints that are applied to generated messages. The constraints include specifying which message parts within the generated message must be digitally signed, and the message parts to attach digitally signed Nonce and timestamp elements to. The following signature and related signature part information are configured:

Table 188. Signature parts information. Use the signature parts to secure messages.

signature parts	Description
keyword	Adds a signature part using keywords. Use the following keywords for the signature parts: • ADDRESSING_HEADERS • BODY • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds a signature part by using an XPath expression.
part	Adds a WSSSignPart object as a target of the signature part.
timestamp	Adds a WSSTimestamp object as a target of the signature part. When specified, the timestamp information also specifies when the message is generated and when it expires.

Table 188. Signature parts information (continued). Use the signature parts to secure messages.

signature parts	Description
header	Adds the header, specified by QName, as a target of the signature part.
securityToken	Adds a SecurityToken object as a target of the signature part.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignature API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing method, the canonicalization method, the security token references, and the signature parts.

Table 189. Signature default behaviors. Several signature behaviors are configured by default.

Signature decisions	Default behavior
Which keywords to use	Sets the keywords. WebSphere Application Server supports the following keywords by default: • ADDRESSING_HEADERS • BODY • TIMESTAMP
Which signature method to use	Sets the signature algorithm. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods: • WSSSignature.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1 • WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1 The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
Which canonicalization method to use	Sets the canonicalization algorithm. The default canonicalization method is EXC C14N. WebSphere Application Server supports the following pre-configured canonicalization methods: • WSSSignature.EXC_C14N; http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignature.C14N: http://www.w3.org/2001/10/xml-c14n#
Whether signature confirmation is required	Sets whether to require signature confirmation. The default value is **false**. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. If required, the value of your signature confirmation is stored in order to use it to validate the signature confirmation after receiving back the message that generated the signature confirmation in the response message. This method is for the requestor side.
Which security token to use	Sets the SecurityToken. The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type. WebSphere Application Server provides the following pre-configured consumer token types: • Derived Key Token • X509 tokens You can also create custom token types, as needed.
Which token reference to set	Sets the refType. SecurityToken.REF_STR is the default value for the type of token reference. WebSphere Application Server supports these pre-configured token references types: • SecurityToken.REF_STR • SecurityToken.REF_KEYID • SecurityToken.REF_EMBEDDED • SecurityToken.REF_THUMBPRINT

If WSSSignature.requireSignatureConfirmation() is called, then the WSSSignature API expects that the response message will include the signature confirmation.

Procedure

1. To configure the signing information in a SOAP message by using the WSS API, first ensure that the application server is installed.
2. Use the WSSSignature API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance. WSSGenerationContext must be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.

d. Creates WSSSignature from the WSSFactory instance using the SecurityToken. The default behavior of WSSSignature is to sign these signature parts: BODY, ADDRESSING_HEADERS, and TIMESTAMP.
e. Adds the part to be signed, if the default part is not appropriate. If the digest method or transform method is changed, creates WSSSignPart and add it to WSSSignature.
f. Creates WSSSignaturePart to WSSSignature. Calls the requiredSignatureConfirmation() method, if the signature confirmation is to be applied.
g. Sets the canonicalization method, if the default is not appropriate.
h. Sets the signature method, if the default is not appropriate.
i. Sets the token reference, if the default is not appropriate.
j. Adds WSSSignature to WSSGenerationContext.
k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signature for the generator section of the bindings. If there is an error condition when signing the message parts, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses methods that are defined in the WSSignature API.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the com.ibm.websphere.wssecurity.wssapi.WSSFactory instance  (step: a)
   WSSFactory factory = com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();

// Generate the WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate the callback handler
   X509GenerateCallbackHandler callbackHandler = new
       X509GenerateCallbackHandler(
       "",
       "dsig-sender.ks",
       "jks",
       "client".toCharArray(),
       "soaprequester",
       "client".toCharArray(),
       "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token to be used for the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class,
        callbackHandler);

// Generate the WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part to be signed  (step: e)
// DEFAULT: WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS,
//          and WSSSignature.TIMESTAMP.

// Set the part in the SOAP Header specified by QName  (step: e)
      sig.addSignHeader(new
                        QName("http://www.w3.org/2005/08/addressing",
                        "MessageID"));

// Set the part specified by the keyword  (step: e)
      sig.addSignPart(WSSSignature.BODY);

// Set the part specified by SecurityToken  (step: e)
   UNTGenerateCallbackHandler untCallbackHandler = new
      UNTGenerateCallbackHandler("Chris", "sirhC");
      SecurityToken unt = factory.newSecurityToken(UsernameToken.class,
         untCallbackHandler);
      sig.addSignPart(unt);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();
      sigPart.setSignPart(WSSSignature.TIMESTAMP);
      sigPart.setDigestMethod(WSSSignPart.SHA256);
      sig.addSignPart(sigPart);
```

```
// Set the part specified by WSSTimestamp  (step: e)
   WSSTimestamp timestamp = factory.newWSSTimestamp();
      sig.addSignPart(timestamp);

// Set the part specified by XPath expression  (step: e)
   StringBuffer sb = new StringBuffer();
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Envelope']");
      sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
         and local-name()='Body']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Ping']");
      sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
         and local-name()='Text']");
      sig.addSignPartByXPath(sb.toString());

// Set to apply the signature confirmation  (step: f)
      sig.requireSignatureConfirmation();

// Set the canonicalization method  (step: g)
// DEFAULT: WSSSignature.EXC_C14N
      sig.setCanonicalizationMethod(WSSSignature.C14N);

// Set the signature method  (step: h)
// DEFAULT: WSSSignature.RSA_SHA1
      sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

// Set the token reference  (step: i)
// DEFAULT: SecurityToken.REF_STR
      sig.setTokenReference(SecurityToken.REF_KEYID);

// Add the WSSSignature to WSSGenerationContext  (step: j)
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgctx);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Next, chose the algorithm methods if you want a method that is different from the default values. If the algorithm methods do not need to be changed, next use the WSSVerification API to verify the signature and specify the algorithm methods in the consumer section of the binding. Note that the WSSVerification API is only supported on the response consumer (client side).

Adding signed parts using the WSSSignPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure parts to be signed for the request generator (client side) bindings, use the WSSSignPart API to protect the integrity of messages and to configure the digest and transform algorithm methods. The WSSSignPart API is part of the com.ibm.websphere.wssecurity.wssapi.signature package.

Before you begin

Either you can use the WSS API or you can configure the policy sets by using the administrative console to configure the signing information. To secure SOAP messages using the signing information, you must complete one of the following tasks:

- Configure the signature information
- Configure signed parts, as needed.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing digest and transform algorithms (for example, SHA1 or TRANSFORM_EXC_C14N).

The signing information specifies the integrity constraints that are applied to generated messages. The signed parts are used to protect the integrity of messages. You can specify the signed parts to add for message integrity protection.

The following table shows the required signed parts when the digital signature security constraint (integrity) is defined:

Table 190. Signed parts information. Use the signed parts to secure messages.

Signed parts	Description
keyword	Adds signed parts using keywords. WebSphere Application Server supports the following keywords for signed parts: • BODY • ADDRESSING_HEADERS • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds the required signed parts by using an XPath expression.
header	Adds the header, specified by QName, as a signed part.
timestamp	Adds a WSSTimestamp object as a signed part. If specified, the timestamp information specifies when the message is generated and when it expires.

Different message parts can be specified in the message protection for request on the generator side. WSSSignPart allows for adding a transform algorithm, setting a digest method, setting objects as targets, specifying whether an element, and the signed parts, such as: the SOAP body, the WS-Addressing header, and timestamp information.

For signing information, certain default behaviors occur. The simplest way to use the WSSSignPart API is to use the default behavior (see the example code). The signed parts default behaviors include:

Table 191. Default behavior of signed parts. Several signed part characteristics are configured by default.

Signature decisions	Default behavior
Which SOAP message parts to sign	WebSphere Application Server supports the following SOAP message parts to be signed and used for message protection: • WSSSignature.BODY • WSSSignature.ADDRESSING_HEADERS • WSSSignature.TIMESTAMP
Which digest method to use	Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. WebSphere Application Server supports the following pre-configured digest methods: • WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1 • WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256 • WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
Which transform algorithms to use	Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. WebSphere Application Server supports the following pre-configured transform algorithms: • WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n# • WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2 Use this transform method to ensure compliance with the Basic Security Profile (BSP). • WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform • WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

Procedure

1. To enable Web Services Security by using the WSS API (WSSSignPart), first ensure that the application server is installed.

2. Use the WSSSignPart API to sign the message parts and specify the algorithms in a SOAP message. The WSS API process for signed parts follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSGenerationContext instance from the WSSFactory instance.
   c. Creates the SecurityToken from WSSFactory to configure the key for signing.
   d. Creates WSSSignature from the WSSFactory instance using the SecurityToken.
   e. Creates WSSSignPart from the WSSFactory instance.
   f. Sets the part to be signed and the digest method or transform method specified by step g or step h if the default is not appropriate.
   g. Sets the digest method if the default is not appropriate.
   h. Sets the transform method if the default is not appropriate.
   i. Adds WSSSignPart to WSSSignature. After any WSSSignPart is set to WSSSignature, the default parts to be signed, which are specified in WSSSignature, are ignored.
   j. Adds WSSSignature to WSSGenerationContext.
   k. Calls WSSGenerationContext.process() with the SOAPMessageContext.

Results

You have completed the steps to configure the signed parts for the generator section of the bindings files. If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses all of methods that are defined in the WSSSignPart API:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance  (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate WSSGenerationContext instance  (step: b)
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

// Generate callback handler
   X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler
      "",
      "dsig-sender.ks",
      "jks",
      "client".toCharArray(),
      "soaprequester",
      "client".toCharArray(),
      "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

// Generate the security token used to the signature  (step: c)
   SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

// Generate WSSSignature instance (step: d)
   WSSSignature sig = factory.newWSSSignature(token);

// Set the part specified by WSSSignPart  (step: e)
   WSSSignPart sigPart = factory.newWSSSignPart();

// Set the part specified by WSSSignPart  (step: f)
   sigPart.setSignPart(WSSSignature.BODY);

// Set the digest method specified by WSSSignPart  (step: g)
   sigPart.setDigestMethod(WSSSignPart.SHA256);

// Set the transform method specified by WSSSignPart  (step: h)
   sigPart.setTransformMethod(WSSSignPart.TRANSFORM_STRT10);

// Add the part specified by WSSSignPart  (step: i)
   sig.addSignPart(sigPart);

// Add the WSSSignature to the WSSGenerationContext  (step: j)
```

```
   gencont.add(sig);

// Generate the WS-Security header  (step: k)
gencont.process(msgcontext);
```

Note: The X509GenerationCallbackHandler needs the key password because the private key is used for signing.

What to do next

Use the WSSVerifyPart API or configure policy sets using the administrative console to verify the signed parts on the consumer side.

Configuring request signing methods for the client:

Use the WSSSignature and WSSSignPart APIs to choose the signing methods. The request signing methods include the signature, canonicalization, digest, and transform methods.

Before you begin

First, you must have specified which parts of the message sent by the client must be digitally signed using the WSS APIs or configuring policy sets using the administrative console.

About this task

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 192. Signing methods. Use the signing methods to secure messages.

Name of method	Description
Canonicalization algorithm	Canonicalizes the <SignedInfo> element before the information is digested as part of the signature operation.
Signature algorithm	Calculates the signature value of the canonicalized <SignedInfo> element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Transform method	Transforms the parts to be signed before the information is digested as part of the signature operation.
Digest method	Calculates the digest value of the transformed parts. The algorithm selected for the client request sender configuration must match the algorithms selected in the server request receiver configuration.

You can use the WSS APIs or configure policy sets using the administrative console to configure the signing algorithm methods. If using the WSS APIs, use the WSSSignature and WSSSignPart APIs to specify which message parts to digitally sign when configuring the client for request signing.

The WSSSignature and WSSSignPart APIs complete the following steps to configure the signature and signed part algorithm methods:

Procedure

1. For the generator binding, the WSSSignature API specifies the signature method. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSSignature.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSSignature.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 digital signature method, http://www.w3.org/2000/09/xmldsig#dsa-sha1.
2. For the generator binding, the WSSSignature API specifies the canonicalization method. WebSphere Application Server supports the following pre-configured canonicalization algorithms:

- WSSSignature.EXC_C14N (the default value): The exclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-exc-c14n#
- WSSSignature.C14N: The inclusive canonicalization algorithm, http://www.w3.org/2001/10/xml-c14n#

3. For the generator binding, the WSSSignPart API specifies the digest method. WebSphere Application Server supports the following pre-configured digest methods:
   - WSSSignPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSSignPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSSignPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the generator binding, the WSSSignPart API specifies the transform method. WebSphere Application Server supports the following pre-configured transform algorithms:
   - WSSSignPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSSignPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSSignPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:
   - http://www.w3.org/TR/1999/REC-xpath-19991116
   - http://www.w3.org/2002/07/decrypt#XML

Results

Using the WSS APIs, you have specified which algorithm methods are used to digitally sign a message when the client sends a message to a server.

Example

The following example is sample code for specifying the signature information, HMAC_SHA1 as signature method, C14N as a canonicalizaion method, SHA256 as a digest method, and EXC_C14N and TRANSFORM_STRT10 as the transform methods:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);
```

```
//set the part specified by WSSSignPart
WSSSignPart sigPart = factory.newWSSSignPart();

//set the digest method
 // DEFAULT: WSSSignPart.SHA1
sigPart.setDigestMethod(WSSSignPart.SHA256);

//add the transform method
// DEFAULT: WSSSignPart.TRANSFORM_EXC_C14N
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_EXC_C14N);
sigPart.addTransformMethod(WSSSignPart.TRANSFORM_STRT10);

 // add the WSSSignPart to the WSSSignature
 sig.addSignPart(sigPart);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

What to do next

After you configure the client to digitally sign the message and to choose the algorithm methods, you must configure the server to verify the digital signature for request signing and to choose the algorithm methods.

Configure policy sets using the administrative console to configure the signature verification information and methods on the server.

Digital signing methods using the WSSSignature API:

You can configure the signing information for the generator binding using the WSS API. To configure the client for request signing, choose the digital signing methods. The algorithm methods include the signing and canonicalization methods.

You must configure generator signing information to protect message integrity by digitally signing SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message.

Methods

Methods that are used for the signing information include the:

Signature method
Sets the signature algorithm method.

Canonicalization method
Sets the canonicalization algorithm method.

Signature algorithms

The signature algorithms specify the algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature method. WebSphere Application Server supports the following pre-configured algorithms:

Table 193. Signature algorithms. The algorithms include the signing methods.

Algorithm	Description
WSSSignature.HMAC_SHA1	A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1
WSSSignature.RSA_SHA1 (the default value)	A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1

For the WSS APIs, WebSphere Application Server does not support the DSA-SHA1 algorithm, http://www.w3.org/2000/09/xmldsig#dsa-sha1

The signing algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

Table 194. Signature canonicalization algorithms. The algorithms include the canonicalization methods.

Algorithm	Description
WSSSignature.EXC_C14N (the default value)	A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSSignature.C14N	A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#

The canonicalization algorithm that is specified for the request generator configuration must match the algorithm that is specified for the request consumer configuration.

The following example provides sample WSS API code that specifies the HMAC_SHA1 as a signature method and C14n as a canonicalization method:

```
//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
     X509GenerateCallbackHandler(
  "",
  "dsig-sender.ks",
  "jks",
  "client".toCharArray(),
  "soaprequester",
  "client".toCharArray(),
  "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
  null);

//generate the security token used to the signature
SecurityToken token = factory.newSecurityToken(X509Token.class,
     callbackHandler);

//generate WSSSignature instance
WSSSignature sig = factory.newWSSSignature(token);

//set the canonicalization method
// DEFAULT: WSSSignature.EXC_C14N
sig.setCanonicalizationMethod(WSSSignature.C14N);

//set the signature method
// DEFAULT: WSSSignature.RSA_SHA1
sig.setSignatureMethod(WSSSignature.HMAC_SHA1);

//add the WSSSignature to the WSSGenerationContext
gencont.add(sig);

//generate the WS-Security header
gencont.process(msgcontext);
```

Signed parts methods using the WSSSignPart API:

You can configure the signed parts information for the generator binding using the WSS API. The algorithms include the digest and transform methods.

You can protect message integrity by configuring signed parts and key information. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signed parts include the:

Digest method
Sets the digest algorithm method.

Transform algorithm
Sets the transform algorithm method.

Digest algorithms

The digest method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 195. Signed parts digest methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.SHA1 (the default value)	A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1
WSSSignPart.SHA256	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256
WSSSignPart.SHA512	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512

Transform algorithms

The transform method algorithm specified within the element is used in the element. WebSphere Application Server supports the following pre-configured algorithms:

Table 196. Signed parts transform methods. The methods are used for the signed parts.

Digest method	Description
WSSSignPart.TRANSFORM_ENVELOPED_SIGNATURE	A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature
WSSSignPart.TRANSFORM_STRT10	A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
WSSSignPart.TRANSFORM_EXC_C14N (the default value)	A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSSignPart.TRANSFORM_XPATH2_FILTER	A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part.

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code for specifying the signature and signed parts, setting the signing key and adding the STR-Transform transform algorithm as signed parts:

```
//get the message context
Object msgcontext = getMessageContext();

//generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

//generate WSSGenerationContext instance
WSSGenerationContext gencont = factory.newWSSGenerationContext();

//generate callback handler
X509GenerateCallbackHandler callbackHandler = new
      X509GenerateCallbackHandler(
```

```
    "",
    "dsig-sender.ks",
    "jks",
    "client".toCharArray(),
    "soaprequester",
    "client".toCharArray(),
    "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP",
    null);

  //generate the security token used to the signature
  SecurityToken token = factory.newSecurityToken(X509Token.class,
       callbackHandler);

  //generate WSSSignature instance
  WSSSignature sig = factory.newWSSSignature(token);

  //set the part specified by WSSSignPart
  WSSSignPart sigPart = factory.newWSSSignPart();

  //set the part specified by WSSSignPart
  sigPart.setSignPart(WSSSignature.BODY);

  //set the digest method specified by WSSSignPart
  sigPart.setDigestMethod(WSSSignPart.SHA256);

  //set the transform method specified by WSSSignPart
  sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);

  //set the part specified by WSSSignPart
  sig.addSignPart(sigPart);

  //add the WSSSignature to the WSSGenerationContext
  gencont.add(sig);

  //generate the WS-Security header
  gencont.process(msgcontext);
```

Attaching the generator token using WSS APIs to protect message authenticity:

When you specify the token generator, the information is used on the generator side to generate the security token.

Before you begin

The token processing and pluggable token architecture in the Web Services Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Services Security run time.

Restriction: The `com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

Note that the key name (KeyName) element is not supported in the application server because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification.

About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web Services Security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Services Security run time.

Note: WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation
- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the generator side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching consumer security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the generator-side security token, use the appropriate pre-configured token generator interface from the WSS APIs to complete the following token configuration process steps:

Procedure

1. Generate the wssFactory instance.
2. Generate the wssGenerationContext instance.

   The WSSGenerationContext interface stores the components for generating Web Services Security (WS-Security), such as the signing and encryption information, the security token, and the time stamp. When the generate() method is called, all of these components are generated.
3. Create the generator-side components, such as the WSSSignature and the WSSEncryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.

5. Specifiy a token generator class name. The token generator class name specifies the required information to generate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specifiy the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   The callback handler implementation obtains the required security token and passes it to the token generator. The token generator inserts the security token in the Web Services Security header within the SOAP message. Also, the token generator is a plug-in point for the pluggable security token framework. Service providers can provide their own implementation, but the implementation must use the WSSGenerationContext interface.

   WebSphere Application Server provides the following default callback handler implementations for the generator side:

   com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback
   This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGUIPromptCallbackHandler
   This class is a callback handler for the Username token with the GUI prompt on the generator side. This instance is used to set the WSSGenerationContext object to generate a Username token.

   com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler
   This class is a callback handler for the Username token on the generator side. This instance is used to set into WSSGenerationContext object to attach a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
   This class is a callback handler that is used to generate the X.509 certificate that is inserted in the Web Services Security header within the SOAP message as a binary security token on the generator side. This instance is used to generate the WSSSignature and WSSEncryption objects, set the objects into the WSSGenerationContext object to generate the X.509 binary security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must be provided on the generator side.

   com.ibm.websphere.wssecurity.callbackhandler.LTPAGenerateCallbackHandler
   This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the generator side. This instance is used to generate WSSSignature object and WSSEncryption object to generate a LTPA token.

   This callback handler is used to validate the LTPA security token inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

   com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
   This class is a callback handler for the Kerberos v5 token on the generator side. This instance is used to set the WSSGenerationContext object to generate the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSSignature and WSSEncryption objects to use the Kerberos session key or derived key in the SOAP message signature and encryption.
7. If a X.509 token is specified, additional token information is also specified.

Table 197. Information for X.509 token. Use the X.509 token for signing and encryption.

Token Information	Description
storeRef	The reference name of the keystore.
storePath	The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations.
storePassword	The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server.
storeType	The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection: **JKS** Use this option if the keystore uses the Java Keystore (JKS) format. **JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption. **JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only). **PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might encrypt keys that use cryptographic hardware to ensure protection. **PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format.
alias	The key alias name. The key alias is used by the key locator to find the key within the keystore file.
keyPassword	The key password that is used for recovering the key. This password is needed to access the key object within the keystore file.
keyName	The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`.
certStores	A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). This step configures a collection certificate store and certificate revocation lists for the generator bindings.
identityAssertion	Specifies whether identity assertion is used. Selects this item if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For an X.509 token generator, the application server sends the original signer certification only.
requestorCertificate	Specifies whether the certificate of the requestor is used.

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain.

d. With keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3

Specifies an X.509 certificate token.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1

Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7

Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2

For an LTPA token, you can use `LTPA` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2

For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes `http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token generator class name, the following token information can be specified:
   a. Whether to use IdentityAssertion option. This option is selected if identity assertion is defined. This option indicates that only the identity of the initial sender is required and inserted into the Web Services Security header within the SOAP message. For example, WebSphere Application Server sends only the user name of the original caller for a Username token generator.
   b. Whether to use RunAsSubject identity option. This option is used if an identity assertion is defined and you want to use the Run As identity instead of the initial caller identity for identity assertion in a downstream call. This option is valid only if you have configured the Username token as the token generator.
   c. Whether to use sendRealm.
   d. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the generated token type is a Username token, and it is available only for the request generator binding.
   e. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   f. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the generated token, and represents the token type of this class:

URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken

Specifies a Username token.

9. If the Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

Token Information	Description	Default Value
name	Kerberos client principal name	
password	Kerberos client password	
realm	Kerberos realm associated with the Kerberos client	Default realm name in Kerberos configuration file. Specify null to use the default value.
targetService	Kerberos service name associated with the target web services.	
targetHost	Kerberos realm name associated with the Kerberos service name.	
tokenValueType	Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification.	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
targetRealm	Kerberos realm name associated with the Kerberos service name.	Default realm name in the Kerberos configuration file
prompt	A boolean value to enable the login prompt.	false
supportTokenRequireSHA1	A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token.	false SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed.
alwaysAPREQ	A boolean value to indicate that the client should always send the Kerberos AP_REQ token in the request messages.	false The SHA1 key is used instead in the subsequent messages. If set to true, the Kerberos AP_REQ token is always used.
requireDKT	A boolean value to require a derived key for message protection.	false
clabel	The client label for the derived key.	WS-SecureConversation Specify null to use the default value.
slabel	The service label for the derived key.	WS-SecureConversation Specify null to use the default value.
keylen	The length of the derived key.	16 Specify zero to use the default value
noncelen	The length of the nonce.	16 Specify zero to use the default value

Token Information	Description	Default Value
encComponent	An instance of WSSEncryption.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.
sigComponent	An instance of WSSSignature.	Set encComponent and sigComponent to null to initialize this first for either the encryption or signature component. Then, use the initialized component only in the callback handler constructor for the second component.

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If Secure Conversation is used for message protection, the following information must be specified:

Information	Description
bootstrapWSSGenerationContext	The bootstrap configuration used to secure the RequestSecurityToken (RST) token.
bootstrapWSSConmingContext	The bootstrap configuration used for consuming a secured RequestSecurityTokenResponse (RSTR).
ENDPOINT_URL	The service end point URL.
EncryptionAlgorithm	This determines the key size.
cLabel	The client label used when creating the derived key.
sLabel	The server label used when creating the derived key.

11. Set the components into the wssGenerationContext object.
12. Invoke the wssGenerationContext.process() method.

Results

Using the Web Services Security API (WSS API) process, you can configured the token generator.

What to do next

Next, you must specify a similar token consumer configuration.

Configuring generator security tokens using the WSS API:

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the generator side, use the Web Services Security APIs (WSS API). The generator security tokens are part of the `com.ibm.websphere.wssecurity.wssapi.token` interface package.

Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security runtime and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must complete the following token tasks:

- Configure the generator tokens.
- Configure the consumer tokens.

About this task

The JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token on the generator side.

On the generator side, the token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Services Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Services Security runtime to the LoginModule. After the token is authenticated, a security token object is created, and the token is passed it to the Web Services Security runtime.

When using the WSS API for generator token creation, certain default behaviors occur. The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide default values for the token type, the token value, and the JAAS confirmation name. The default token behaviors include:

Table 198. Token decisions and default behaviors. Several token characteristics are configured by default.

Generator token decisions	Default behavior
Which token type to use	The token type specifies which type of token to use for message integrity, message confidentiality, or message authenticity. WebSphere Application Server provides the following pre-configured generator token types for message integrity and message confidentiality: • Derived key token • X509 tokens You can also create custom token types, as needed. WebSphere Application Server also provides the following pre-configured generator token types for the message authenticity: • Username token • LTPA tokens • X509 tokens You can also create custom token types, as needed.
What JAAS login configuration name to specify	The JAAS login configuration name specifies which JAAS login configuration name to use.
Which configuration type to use	The JAAS login module specifies the configuration type. Only the pre-configured generator configuration types can be used for generator token types.

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, the XML format, and the cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following tokens types are subclasses of the generic security token class:

Table 199. Subclasses of the SecurityToken. Use the subclasses to represent the security token.

Token type	JAAS login configuration name
Username token	system.wss.generate.unt
Security context token	system.wss.generate.sct
Derived key token	system.wss.generate.dkt

The following tokens types are subclasses of the binary security token class:

Table 200. Subclasses of the BinarySecurityToken. Use the subclasses to represent the binary security token.

Token type	JAAS login configuration name
LTPA token	system.wss.generate.ltpa
LTPA propagation token	system.wss.generate.ltpaProp
X.509 token	system.wss.generate.x509
X.509 PKI Path token	system.wss.generate.pkiPath
X.509 PKCS7 token	system.wss.generate.pkcs7

Note:

- For each JAAS login token generator configuration name, there is a respective token consumer configuration name. For example, for the Username token, the respective token consumer configuration name is system.wss.consume.unt.
- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

Procedure

1. To configure the securityToken package, `com.ibm.websphere.wssecurity.wssapi.token`, first ensure that the application server is installed.
2. Use the Web Services Security token generator process to configure the tokens. For each token type, the process is similar to the following process that demonstrates the UsernameToken token generator process:
   a. Use WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Create the WSSGenerationContext instance from the WSSFactory instance.
   c. Create a JAAS CallbackHandler. The authentication data, such as the user name and password are specified as part of the CallbackHandler. For example, the following code specifies Chris as the user name and sirhC as the password: UNTGenerationCallbackHandler("Chris", "sirhC");
   d. Call any JAAS CallbackHandler parameters and review the token class information for which parameters are required or optional. For example, for the UsernameToken, the following parameters can be configured also:

      Nonce

      Indicates whether a nonce is included in the user name token for the token generator. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of user name tokens. The nonce value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

Created timestamp

Indicates whether to insert a time stamp into the UsernameToken. The timestamp value is valid only when the generated token type is a UsernameToken and only when it applies to the request generator binding.

e. Create the SecurityToken from WSSFactory.

   By default, the UsernameToken API specifies the ValueType as: "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"

   By default, the UsernameToken API provides the QName of this class and specifies the NamespaceURI as http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd and also specifies the LocalPart as UsernameToken.

f. Optional: Specify the JAAS login module configuration name. On the generator side, the configuration type is always `generate` (for example, `system.wss.generate.unt`).

g. Add the SecurityToken to the WSSGenerationContext.

h. Call WSSGenerationContext.process() and generate the WS-Security header.

Results

If there is an error condition, a WSSException is provided. If successful, the WSSGenerationContext process() is called, and the security token for the generator binding is attached.

Example

The following example code shows how to use WSS APIs to create a Username security token, attach the Username token to the SOAP message, and configure the Username token in the generator binding.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

   // generate WSSGenerationContext instance
   WSSGenerationContext gencont = factory.newWSSGenerationContext();

   // generate callback handler
   UNTGenerateCallbackHandler untCallbackHandler =
   new UNTGenerateCallbackHandler("Chris", "sirhC");

   // generate the username token
   SecurityToken unt = factory.newSecurityToken(UsernameToken.class, untCallbackHandler);

   // add the SecurityToken to the WSSGenerationContext
   gencont.add(unt);

   // generate the WS-Security header
   gencont.process(reqContext);
```

The following example code shows how to modify the preceding Username token sample to create an LTPAv2 token from the runAs identity on the current thread. The two lines of code that instantiate the callback handler and create the security token are replaced with the following two lines of code:

```
// generate callback handler
LTPAGenerateCallbackHandler ltpaCallbackHandler = new LTPAGenerateCallbackHandler(null, null);

// generate the LTPAv2 token
SecurityToken ltpa = wssfactory.newSecurityToken(LTPAv2Token.class, ltpaCallbackHandler);
```

The instantiation of the LTPAGenerateCallbackHandler object with (null, null) indicates that the LTPA token should be generated from the current runAs identity. If the callback handler is instantiated with basicAuth information, ("userName", "password"), a new LTPA token is created using the specified basicAuth information.

The following example shows how to use secure conversation with the WSS APIs to configure the generator tokens, as well as the consumer tokens. In this example, the SecurityContextToken token is created using the WS-SecureConversation draft namespace: `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`. To use the WS-SecureConversation version 1.3 namespace, `http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct`, specify SecurityContextToken13.class instead of SecurityContextToken.class.

```
// import the packages
import javax.xml.ws.BindingProvider;
import com.ibm.websphere.wssecurity.wssapi.*;
import com.ibm.websphere.wssecurity.callbackhandler.*;
...
   // obtain the binding provider
   BindingProvider bp = ... ;

   // get the request context
   Map<String, Object> reqContext = bp.getRequestContext();

   // generate WSSFactory instance
   WSSFactory wssFactory = WSSFactory.getInstance();

   WSSGenerationContext bootstrapGenCon = wssFactory.newWSSGenerationContext();

   // Create a Timestamp
   ...
   // Add Timestamp
   ...

   // Sign the SOAP Body, WS-Addressing headers, and Timestamp
   X509GenerateCallbackHandler btspReqSigCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqSigToken = wssFactory.newSecurityToken(X509Token.class,
                                                               btspReqSigCbHandler);
   WSSSignature bootstrapReqSig = wssFactory.newWSSSignature(btspReqSigToken);
   bootstrapReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);

   // Add Sign Parts
   ...
   bootstrapGenCon.add(bootstrapReqSig);

   // Encrypt the SOAP Body and the Signature
   X509GenerateCallbackHandler btspReqEncCbHandler = new X509GenerateCallbackHandler(...);
   SecurityToken btspReqEncToken = wssFactory.newSecurityToken(X509Token.class,
                                                               btspReqEncCbHandler);
   WSSEncryption bootstrapReqEnc = wssFactory.newWSSEncryption(btspReqEncToken);
   bootstrapReqEnc.setEncryptionMethod(WSSEncryption.AES128);
   bootstrapReqEnc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);

   // Add Encryption parts
   ...
   bootstrapGenCon.add(bootstrapReqEnc);
   WSSConsumingContext bootstrapConCon = wssFactory.newWSSConsumingContext();
   X509ConsumeCallbackHandler btspRspVfyCbHandler = new X509ConsumeCallbackHandler(....);
   WSSVerification bootstrapRspVfy = wssFactory.newWSSVerification(X509Token.class,
                                                                   btspRspVfyCbHandler);
   bootstrapRspVfy.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

   // Add Verify parts
   ...
   bootstrapConCon.add(bootstrapRspVfy);
   X509ConsumeCallbackHandler btspRspDecCbHandler = new X509ConsumeCallbackHandler(...);
   WSSDecryption bootstrapRspDec = wssFactory.newWSSDecryption(X509Token.class,
                                                               btspRspDecCbHandler);
   bootstrapRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
   bootstrapRspDec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA15);

   // Add Decryption parts
   ...
   bootstrapConCon.add(bootstrapRspDec);
   SCTGenerateCallbackHandler sctgch = new SCTGenerateCallbackHandler(bootstrapGenCon,
                                                                      bootstrapConCon,
                                                                      ENDPOINT_URL,
                                                                      WSSEncryption.AES128);
   SecurityToken[] scts = wssFactory.newSecurityTokens(new Class[]{SecurityContextToken.class},
                                                       sctgch);
   SecurityContextToken sct = (SecurityContextToken)scts[0];

   // Use the SCT to generate DKTs for Secure Conversation
   // Signature algorithm and client and service labels
   DerivedKeyToken dktSig = sct.getDerivedKeyToken(WSSSignature.HMAC_SHA1,
                                                   "WS-SecureConversation",
                                                   "WS-SecureConversation");

   // Encryption algorithm and client and service labels
   DerivedKeyToken dktEnc = sct.getDerivedKeyToken(WSSEncryption.AES128,
                                                   "WS-SecureConversation",
                                                   "WS-SecureConversation");
```

```
// Create the application generation context for the request message
WSSGenerationContext applicationGenCon = wssFactory.newWSSGenerationContext();

// Create and add Timestamp
...

// Add the derived key token and Sign the SOAP Body and WS-Addressing headers
WSSSignature appReqSig = wssFactory.newWSSSignature(dktSig);
appReqSig.setSignatureMethod(WSSSignature.HMAC_SHA1);
appReqSig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
...
applicationGenCon.add(appReqSig);

// Add the derived key token and Encrypt the SOAP Body and the Signature
WSSEncryption appReqEnc = wssFactory.newWSSEncryption(dktEnc);
appReqEnc.setEncryptionMethod(WSSEncryption.AES128);
appReqEnc.setTokenReference(SecurityToken.REF_STR);
appReqEnc.encryptKey(false);
...
applicationGenCon.add(appReqEnc);

// Create the application consuming context for the response message
WSSConsumingContext applicationConCon = wssFactory.newWSSConsumingContext();

//client and service labels and decryption algorithm
SCTConsumeCallbackHandler sctCbHandler = new SCTConsumeCallbackHandler("WS-SecureConversation",
                                                                       "WS-SecureConversation",
                                                                       WSSDecryption.AES128);

// Derive the token from SCT and use it to Decrypt the SOAP Body and the Signature
WSSDecryption appRspDec = wssFactory.newWSSDecryption(SecurityContextToken.class,
                                                      sctCbHandler);
appRspDec.addAllowedEncryptionMethod(WSSDecryption.AES128);
appRspDec.encryptKey(false);
...
applicationConCon.add(appRspDec);

// Derive the token from SCT and use it to Verify the
// signature on the SOAP Body, WS-Addressing headers, and Timestamp
WSSVerification appRspVfy = wssFactory.newWSSVerification(SecurityContextToken.class,
                                                          sctCbHandler);
...
applicationConCon.add(appRspVfy);
...
applicationGenCon.process(reqContext);
applicationConCon.process(reqContext);
```

What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar consumer tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by signing the SOAP message or by encrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

Sending self-issued SAML bearer tokens using WSS APIs:

You can create self-issued SAML tokens with the bearer subject confirmation method and then send these tokens with Web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings.

About this task

You can build your web services client to use SAML tokens with the bearer subject confirmation method in SOAP request messages using the Web Services Security programming interfaces. Using the programming interfaces in a web services client to specify the use of SAML tokens with bearer subject confirmation is an alternative approach to using policy sets and binding configurations.

You can create a self-issued SAML token and then send the SAML token in web services request messages from a web services client. The web services application client used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code snippets from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

      The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

   Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Attach the SAML20 Bearer WSHTTPS default policy set to the web services provider. This policy set is used to protect messages using HTTPS transport. Read about configuring client and provider bindings for the SAML Bearer token for details on how to attach the SAML20 Bearer WSHTTPS default policy set to the Web services provider. The example in this procedure uses self-issued SAML tokens. When you configure the provider bindings, the truststore configuration and certificate must match the signing key of the self-issued token.

3. Assign the SAML Bearer Provider sample default general bindings to the sample web services provider. Read about configuring client and provider bindings for the SAML bearer token for details on assigning the SAML Bearer Provider sample default general bindings to your web services application.

4. Create the self-issued SAML token. The following code snippet illustrates creating the SAML token:

```
// Create the SAML token.
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the CallService() method to specify the Web services security configuration parameters that are required to invoke a target Web services provider using a self-issued SAML token.

      The CallService() method sets the configuration parameters that are required by the Web Services Security runtime environment via the `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` custom property to generate a self-issued SAMLToken.

Read about configuring a SAML token during token creation for more information about how you can specify configuration properties to control how the token is configured.

b. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.

c. Use the WSSFactory newSecurityToken method to specify how to create the SAML token.

Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

Creating a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the PolicyTool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken
```

The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are creating. This object points to a SAMLGenerateCallbackHandler object that specifies the configuration parameters described in the following table:

Table 201. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the bearer subject confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `Bearer` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Uses the constant value, `WSSConstants.SAML.SAML20_VALUE_TYPE`, to specify a SAML 2.0 token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. In this scenario, specify the token value type by the valueType attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.SAML_NAME_IDENTIFIER	Specifies a user identity such as `myname` as the NameID value in the SAML token. If you do not define this parameter when using the Thin Client for JAX-WS, the NameID value does not contain useful information. If you are using a web services managed client, such a Java Platform, Enterprise Edition (Java EE) application making a web services request invocation, the Web Services Security runtime environment tries to extract user security information from the security context. Similarly, if you do not define this parameter for a managed web services client, the NameID value contains an UNAUTHENTICATED name identifier. This property is not used if your web services client has policy set attachments. Read about sending SAML tokens to learn more about sending the SAML token identity and attributes.	No
SamlConstants.SIGNATURE_REQUIRED	Specifies whether the issuer is required to digitally sign the SAML token. A `true` value specifies that issuer is required to digitally sign the SAML token. This value is the default.	No

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
```

d. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

5. Add the SAML token to the SOAP security header of a Web services request messages.
   a. Initialize the web services client and configure the SOAPAction properties. The following code snippet illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");
```

   b. Initialize the WSSGenerationContext. The following code illustrates the use of the WSSGenerationContext interface to initialize a generation context and enable you to insert the `SAMLToken` into the web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

      Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. Use the PolicyTool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
 "permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

   c. Add the timestamp element in the SOAP messages security header. The SAML20 Bearer WSHTTPS default policy set requires web services requests and response messages to carry a timestamp element in SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call specifies the timestamp to put into a request message:

```
// Add a timestamp to the request message.
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);

gencont.process(requestContext);
```

   d. Attach WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

   e. Specify SSL transport level message protection using JVM properties.

      The SAML20 Bearer WSHTTPS default policy set requires transport-level message protection using SSL. Specify SSL transport-level message protection using the following JVM property:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

      Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

Results

You have created a self-issued SAML token with the bearer subject confirmation method and then sent this token with web services request messages using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a web services client application that demonstrates how to create a self-issued SAML token and send that SAML token in web services request messages. If your usage

scenario requires SAML tokens, but does not require your application to pass the SAML tokens using web services messages, you only need to use the first part of the following sample code, up through the `// Initialize web services client` section.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */

package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

/**
 * SampleClient
 * main entry point for thin client JAR sample
 * and worker class to communicate with the services
 */
public class SampleClient {

  private String urlHost = "localhost";
  private String urlPort = "9443";
  private static final String CONTEXT_BASE = "/WSSampleSei/";
  private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
  private String message = "HELLO";
  private String uriString = "https://" + urlHost + ":" + urlPort;
  private String endpointURL = uriString + ECHO_CONTEXT12;
  private String input = message;

  /**
   * main()
   *
   * see printusage() for command-line arguments
   *
   * @param args
   */
  public static void main(String[] args) {
    SampleClient sample = new SampleClient();
    sample.CallService();
  }

  /**
   * CallService Parms were already read. Now call the service proxy classes
   *
   */
  void CallService() {
    String response = "ERROR!:";
    try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
      System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");

      // Initialize WSSFactory object
      WSSFactory factory = WSSFactory.getInstance();
      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      // Initialize SAML issuer configuration via custom properties
      HashMap<Object, Object> customProps = new HashMap<Object,Object>();

      customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
      customProps.put(SamlConstants.TTL_PROP, "3600000");
      customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
      customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
      customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
      customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
      customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
      gencont.add(customProps); //Add custom properties
```

```
      // Create SAMLToken
      HashMap<Object, Object> map = new HashMap<Object, Object>();
      map.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");
      map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
      map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
      map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
      SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);

      SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

      System.out.println("SAMLToken id = " + samlToken.getId());

      // Initialize web services client
      EchoService12PortProxy echo = new EchoService12PortProxy();
      echo._getDescriptor().setEndpoint(endpointURL);

      // Configure SOAPAction properties
      BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
      Map<String, Object> requestContext = bp.getRequestContext();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
      requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
      requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

      gencont.add(samlToken);

      // Add timestamp
      WSSTimestamp timestamp = factory.newWSSTimestamp();
      gencont.add(timestamp);

      gencont.process(requestContext);

      // Build the input object
      EchoStringInput echoParm =
        new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
      echoParm.setEchoInput(input);
      System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

      // Prepare to consume timestamp in response message.
      WSSConsumingContext concont = factory.newWSSConsumingContext();
      concont.add(WSSConsumingContext.TIMESTAMP);
      concont.process(requestContext);

      // Call the service
      response = echo.echoOperation(echoParm).getEchoResponse();

      System.out.println(">> CLIENT: SEI Echo invocation complete.");
      System.out.println(">> CLIENT: SEI Echo response is: " + response);
    } catch (Exception e) {
      System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
      e.printStackTrace();
    }
  }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _191EBC44865015D9AB1270745072344
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to https://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Inserting SAML attributes using WSS APIs:

You can insert custom attributes into self-issued SAML tokens by using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about propagating self-issued SAML bearer tokens by using WSS APIs.
- Read about propagating self-issued SAML sender-vouches tokens by using WSS APIs with message level protection.

- Read about propagating self-issued SAML sender-vouches tokens by using WSS APIs with SSL transport protection.
- Read about propagating self-issued SAML holder-of-key tokens with symmetric key by using WSS APIs.
- Read about propagating self-issued SAML holder-of-key tokens with asymmetric key by using WSS APIs.

About this task

This task shows example code that inserts custom attributes into self-issued SAML security tokens. This particular example uses the bearer subject confirmation method. You can add attributes to any SAML security tokens, and the same code can be used with other subject confirmation methods.

Procedure

Insert custom attributes when creating SAML security tokens; for example:

```
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;
import com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute;

WSSFactory factory = WSSFactory.getInstance();
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");
map.put(SamlConstants.Token_REQUEST, "issue");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
String groups[] = {"IBMer", "Texan"};
SAMLAttribute sattribute = new SAMLAttribute("Membership", groups, null,null, null, null);
al.add(sattribute);
String gender[] = {"Female"};
sattribute = new SAMLAttribute("Gender", gender, null,null, null, null);
al.add(sattribute);
map.put(SamlConstants.SAML_ATTRIBUTES, al);
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler,
                                                   "system.wss.generate.saml");
```

Results

You have inserted custom attributes to a SAML security token.

Example

The following example shows the custom attributes in the SAML Assertion:

```
<saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
                 Version="2.0"
                      ID="_E62A1CA3C2F21D9A9B1287772824570"
                      IssueInstant="2010-10-22T18:40:24.531Z">
    <saml2:Issuer>example.com</saml2:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    ...
    </ds:Signature>
    <saml2:Subject>
        <saml2:NameID>Alice</saml2:NameID>
        <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer"></saml2:SubjectConfirmation>
    </saml2:Subject>
    <saml2:Conditions NotBefore="2010-10-22T18:40:24.531Z"
                   NotOnOrAfter="2010-10-22T19:40:24.531Z">
    </saml2:Conditions>
    <saml2:AttributeStatement>
        <saml2:Attribute Name="Membership">
            <saml2:AttributeValue>IBMer</saml2:AttributeValue>
            <saml2:AttributeValue>Texan</saml2:AttributeValue>
        </saml2:Attribute>
```

```
        <saml2:Attribute Name="Gender">
            <saml2:AttributeValue>Female</saml2:AttributeValue>
        </saml2:Attribute>
    </saml2:AttributeStatement>
</saml2:Assertion>
```

What to do next

Merge the code with the example code listed in the “Propagating self-issued SAML bearer tokens by using WSS APIs” topic to generate SAML security tokens. You can see SAML attributes in the SAML Assertions.

Sending self-issued SAML sender-vouches tokens using WSS APIs with message level protection:

You can create self-issued SAML tokens with the sender-vouches subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages with message level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings.

About this task

You can protect SOAP request messages and SAML tokens by using the Web Services Security programming interface to satisfy the sender-vouches subject confirmation method validation requirements with message level protection. Using the programming interfaces in web services client is an alternative approach to using policy set and binding configuration.

You can create a self-issued SAML token and then send the SAML token in web services request messages from a web services client. The web services application client used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code snippets from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

This product does not provide a default policy set that requires SAML tokens with sender-vouches subject confirmation method. Read about configuring client and provider bindings for the SAML sender-vouches token to learn more about how to create a Web Services Security policy to require SAML tokens with sender-vouches subject confirmation and how to create a custom binding configuration. You must attach the policy and binding to the web services provider. The code sample described in this task assumes that the web services provider policy requires that both the SAML tokens and the message bodies are digitally signed by using an X.509 security token.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.
   b. Obtain the JaxWSServicesSamples client code.

For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Use the CallService() method to specify the Web services security configuration parameters that are required to invoke a target Web services provider using a self-issued SAML token.

   The CallService() method sets configuration parameters that are required by the Web Services Security runtime environment via the `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` custom property to generate a self-issued SAMLToken.

   The following code snippet illustrates using the CallService() method to set the `SamlConstants.SAML_SELF_ISSUER_CONFIG` system property:

```
public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
}

/**
 * CallService Parms were already read. Now call the service proxy classes
 *
 */
void CallService() {
  String response = "ERROR!:";
  try {
    System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");
```

   Read about configuring a SAML token during token creation for more information about how you can specify configuration properties to control how the token is configured.

3. Add the Thin Client for JAX-WS JAR file to the class path. Add *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.

4. Create the self-issued SAML token. The following code snippet illustrates creating the SAML token:

```
// Create SAMLToken
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");
System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the WSSFactory newSecurityToken method to specify how to create the SAML token.

      Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

      Creating a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken
```

      The `SAMLToken.class` parameter specifies the type of security token to create.

      The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are creating. This object points to a SAMLGenerateCallbackHandler object that specifies the following configuration parameters described in the following table:

Table 202. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Uses the constant value, `WSSConstants.SAML.SAML20_VALUE_TYPE` to specify a SAML 2.0 token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. In this scenario, specify the token value type by the valueType attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.SAML_NAME_IDENTIFIER	Specifies a user identity such as `myname` as the NameID value in the SAMLToken. If you do not define this parameter when using the Thin Client for JAX-WS, the NameID value does not contain useful information. If you are using a web services managed client, such a Java Platform, Enterprise Edition (Java EE) application making a web services request invocation, the Web Services Security runtime environment tries to extract user security information from the security context. Similarly, if you do not define this parameter for a managed web services client, the NameID value contains an UNAUTHENTICATED name identifier. This property is not used if your web services client has policy set attachments. Read about sending SAML tokens to learn more about sending the SAML token identity and attributes.	No
SamlConstants.SIGNATURE_REQUIRED	Specifies whether the issuer is required to digitally sign the SAML token. A `true` value specifies that issuer is required to digitally sign the SAML token. This value is the default.	No

The `system.wss.generate.saml` parameter specifies to use a Java Authentication and Authorization Service (JAAS) login configuration and specifies the login module that is invoked to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
Djava.security.auth.login.config=profile_root/properties/wsjaas.conf
```

Alternatively, you can specify a JAAS login configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

5. Add the SAML token to the SOAP security header of web services request messages.

a. Initialize the web services client and configure the SOAPAction properties. The following code example illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

b. Initialize the WSSGenerationContext. The following code snippet illustrates the use of the gencont.object of the WSSGenerationContext type to initialize a generation context to enable you to insert the SAMLToken into a web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

Specifically, the gencont.add(samlToken) method call specifies to put the SAML token into a request message. This operation requires the client code to have the following Java 2 Security permission:

```
 "permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

6. Add an X.509 token for message protection.

This sample code uses the dsig-sender.ks key file and the SOAPRequester sample key. You must not use the sample key in a production environment. The following code snippet illustrates adding an X.509 token for message protection:

```
// Add an X.509 Token for message protection
X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
        null,
        "profile_root/etc/ws-security/samples/dsig-sender.ks",
        "JKS",
        "client".toCharArray(),
        "soaprequester",
        "client".toCharArray(),
        "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

SecurityToken x509 = factory.newSecurityToken(X509Token.class,
        x509callbackHandler, "system.wss.generate.x509");

WSSSignature sig = factory.newWSSSignature(x509);
sig.setSignatureMethod(WSSSignature.RSA_SHA1);

WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
sig.addSignPart(sigPart);
sig.addSignPart(WSSSignature.BODY);
```

a. Create a WSSSignature object with the X509 token. The following line of code creates a WSSSignature object with the X509 token:

```
WSSSignature sig = factory.newWSSSignature(x509);
```

b. Add the signed part to use for message protection. The following line of code specifies to add WSSSignature.BODY as the signed part:

```
sig.addSignPart(WSSSignature.BODY);
```

c. Add the timestamp element in the SOAP messages security header. The SAML20 SenderVouches WSHTTPS and SAML11 SenderVouches WSHTTPS policy sets require web services requests and response messages to carry a timestamp element in the SOAP messages Security header. In the following code snippet, the factory.newWSSTimestamp() method call generates the timestamp, and the gencont.add(timestamp) method call adds the timestamp into the request message:

```
// Add Timestamp
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);

WSSConsumingContext concont = factory.newWSSConsumingContext();
```

d. Configure the verification of the digital signature in the response message.

A separate WSSSignPart is needed to specify the SecurityTokenReference transformation algorithm that is represented by the WSSSignPart.TRANSFORM_STRT10 attribute. A SAML Token cannot be digitally signed directly. This attribute enables the Web Services Security runtime environment to generate a SecurityTokenReference element to reference the SAMLToken and to digitally sign the SAMLToken using the SecurityTokenReference transformation. The following line of code specifies to use the WSSSignPart.TRANSFORM_STRT10 attribute:

```
WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
```

e. Attach the WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a

request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches the WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

7. Use the X.509 token to validate the digital signature and the integrity of the response message. If the provider policy requires the response message to be digitally signed, you must initialize the X.509 token.
   a. A X509ConsumeCallbackHandler object is initialized with a truststore, dsig-receiver.ks, and a certificate path object to validate the provider digital signature. The following line of code is used to initialize the X509ConsumeCallbackHandler object:

```
X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
        "profile_root/etc/ws-security/samples/dsig-receiver.ks",
        "JKS",
        "server".toCharArray(),
        certList,
        java.security.Security.getProvider("IBMCertPath"));
```

   b. A WSSVerification object is created and the message body is added to the verification object so that the Web Services Security runtime environment validates the digital signature.

      The following line of code is used to initialize the WSSVerification object:

```
WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);
```

      The WSSConsumingContext object now contains all the security information that is required to format a request message. The `concont.process(requestContext)` method call attaches the WSSConsumingContext object to the response method; for example:

```
// Attaches the WSSConsumingContext object to the web services RequestContext object.
concont.process(requestContext);
```

Results

You have created a self-issued SAML token with the sender-vouches confirmation method and then sent this token with web services request messages using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to create a self-issued SAML sender-vouches token and send that SAML token in web services request messages. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSException;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignPart;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignature;
import com.ibm.websphere.wssecurity.wssapi.verification.WSSVerification;
import com.ibm.websphere.wssecurity.wssapi.token.X509Token;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.security.InvalidAlgorithmParameterException;
```

```
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.cert.CertStore;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.CollectionCertStoreParameters;
import java.security.cert.X509Certificate;
import java.util.HashSet;
import java.util.Set;
import java.util.HashMap;
import java.util.Map;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
  private String urlHost = "localhost";
  private String urlPort = "9081";
  private static final String CONTEXT_BASE = "/WSSampleSei/";
  private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
  private String message = "HELLO";
  private String uriString = "http://" + urlHost + ":" + urlPort;
  private String endpointURL = uriString + ECHO_CONTEXT12;
  private String input = message;

  /**
   * main()
   *
   * see printusage() for command-line arguments
   *
   * @param args
   */
  public static void main(String[] args) {
    SampleSamlSVClient sample = new SampleSamlSVClient();
    sample.CallService();
  }

  /**
   * CallService Parms were already read. Now call the service proxy classes.
   *
   */
  void CallService() {
    String response = "ERROR!:";
    try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");

      // Initialize WSSFactory object
      WSSFactory factory = WSSFactory.getInstance();
      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      // Initialize SAML issuer configuration via custom properties
      HashMap<Object, Object> customProps = new HashMap<Object,Object>();

      customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
      customProps.put(SamlConstants.TTL_PROP, "3600000");
      customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
      customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
      customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
      customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
      customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
      gencont.add(customProps); //Add custom properties

      // Create SAMLToken
      HashMap<Object, Object> map = new HashMap<Object, Object>();
      map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
      map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
      map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
      map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
      SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);

      SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

      System.out.println("SAMLToken id = " + samlToken.getId());

      // Initialize web services client.
      EchoService12PortProxy echo = new EchoService12PortProxy();
      echo._getDescriptor().setEndpoint(endpointURL);

      // Configure SOAPAction properties
      BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
      Map<String, Object> requestContext = bp.getRequestContext();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
      requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
      requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
```

```
gencont.add(samlToken);

// Add X.509 Tokens for message protection
X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
    null,
    "profile_root/etc/ws-security/samples/dsig-sender.ks",
    "JKS",
    "client".toCharArray(),
    "soaprequester",
    "client".toCharArray(),
    "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

SecurityToken x509 = factory.newSecurityToken(X509Token.class,
    x509callbackHandler, "system.wss.generate.x509");

WSSSignature sig = factory.newWSSSignature(x509);
sig.setSignatureMethod(WSSSignature.RSA_SHA1);

WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
sig.addSignPart(sigPart);
sig.addSignPart(WSSSignature.BODY);

// Add timestamp
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);

WSSConsumingContext concont = factory.newWSSConsumingContext();

// Prepare to consume timestamp in response message
concont.add(WSSConsumingContext.TIMESTAMP);

// Prepare to verify digital signature in response message
X509Certificate x509cert = null;
try {
  InputStream is = new FileInputStream("profile_root/etc/ws-security/samples/intca2.cer");
  CertificateFactory cf = CertificateFactory.getInstance("X.509");
  x509cert = (X509Certificate) cf.generateCertificate(is);
} catch (FileNotFoundException e1) {
  throw new WSSException(e1);
} catch (CertificateException e2) {
  throw new WSSException(e2);
}
Set<Object> eeCerts = new HashSet<Object>();
eeCerts.add(x509cert);

java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);

CertStore cert = null;
try {
  cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
} catch (NoSuchProviderException e1) {
  throw new WSSException(e1);
} catch (InvalidAlgorithmParameterException e2) {
  throw new WSSException(e2);
} catch (NoSuchAlgorithmException e3) {
  throw new WSSException(e3);
}
if (certList != null) {
  certList.add(cert);
}

X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
    "profile_root/etc/ws-security/samples/dsig-receiver.ks",
    "JKS",
    "server".toCharArray(),
    certList,
    java.security.Security.getProvider("IBMCertPath"));

WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);

ver.addRequiredVerifyPart(WSSVerification.BODY);
concont.add(ver);

gencont.process(requestContext);
concont.process(requestContext);

// Build the input object
EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
echoParm.setEchoInput(input);
System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

// Call the service
response = echo.echoOperation(echoParm).getEchoResponse();
```

```
      System.out.println(">> CLIENT: SEI Echo invocation complete.");
      System.out.println(">> CLIENT: SEI Echo response is: " + response);
    } catch (Exception e) {
      System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
      e.printStackTrace();
    }
  }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9080/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Sending self-issued SAML sender-vouches tokens using WSS APIs with SSL transport protection:

You can create self-issued SAML tokens with the sender-vouches subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages with transport protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, SSL transport protection, and the use of policy sets to configure and administer web services settings.

About this task

You can build your web services client to use SAML tokens with the sender-vouches subject confirmation method in SOAP request messages using the Web Services Security programming interfaces. Using the programming interfaces in a web services client to specify the use of SAML tokens with sender-vouches subject confirmation using message protection at the transport level is an alternative approach to using policy sets and binding configurations.

You can create a self-issued SAML token and then send the SAML token in web services request messages from a web services client. The web services client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to use the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.
   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Create a copy of either the SAML20 Bearer WSHTTPS default policy set or the SAML11 Bearer WSHTTPS default policy set.

   Provide a name for the copy of the policy set; for example SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS to help you identify that this new policy set uses the sender-vouches confirmation method.

   No additional change is required to the new policy file because the subject confirmation method is specified in the binding configuration and not in the policy.

   The new policy file contains either SAMLToken20Bearer or the SAMLToken11Bearer as the policy identifiers. Change the identifier of the SAMLToken20Bearer policy to SAMLToken20SV or change the identifier of the SAMLToken11Bearer policy to SAMLToken11SV to specify a more descriptive name. Changing the identifier of the policy does not change the policy enforcement in any way; however, adding a descriptive identifier helps you to identify that these policy identifiers use the sender-vouches confirmation method.

   If you want to view the settings of these policies, use the administrative console to complete the following actions:

   a. Click **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name***.

   b. Click the **WS-Security** policy in the policies table.

   c. Click the **Main policy** link or the **Bootstrap policy** link.

   d. Click **Request token policies** from the Policy Details section.

3. Attach the new SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS policy set to the web services provider application. Read about configuring client and provider bindings for the SAML sender-vouches token for details on attaching this policy set to your web services provider application.

4. Create a copy of the SAML Bearer Provider sample default general bindings.

   a. For the new copy of the default policy set, provide a name that includes sender-vouches, such as `SAML Sender-vouches provider binding`.

   b. Change the value of the confirmationMethod property to `sender-vouches` in the token consumer configuration for the intended SAML token version. Read about configuring client and provider bindings for the SAML sender-vouches token for details on modifying the sender-vouches bindings to satisfy the vouching requirement.

5. Assign the new provider binding to the JaxWSServicesSamples provider sample. Read about configuring client and provider bindings for the SAML sender-vouches for details on assigning the SAML sender-vouches provider sample, default general bindings to your web services provider application.

6. Enable the web services provider SSL configuration attribute, clientAuthentication, to require X.509 client certificate authentication.

   The clientAuthentication attribute determines whether SSL client authentication is required. To specify the clientAuthentication attribute, use the administrative console to complete the following actions:

   a. Click **Security** > **SSL certificates and key management** > **Manage endpoint security configurations** > **{Inbound | Outbound}** > ***SSL_configuration***.

   b. Click the **WC_defaulthost_secure** link.

   c. Under Related Items, click the **SSL_configurations** link.

d. Select the **NodeDefaultSSLSettings** resource.

e. Click **Quality of protection (QoP) settings** link.

f. Select **Required** from the menu to specify client authentication.

Read about creating a secure sockets layer configuration to learn more about configuring the clientAuthentication attribute.

7. In the web services client code, use the CallService() method to specify the properties file that contains configuration parameters required to generate a self-issued SAML token.

   The CallService() method specifies configuration parameters that are required by the Web Services Security runtime environment to generate a self-issued SAMLToken.

   The following code snippet illustrates using the CallService() method to specify Web services security configuration parameters:

```
public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
 }

/**
 * CallService Parms were already read. Now call the service proxy classes
 *
 */
void CallService() {
  String response = "ERROR!:";
  try {
    System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
    // Initialize WSSFactory object
    WSSFactory factory = WSSFactory.getInstance();

    // Initialize WSSGenerationContext
    WSSGenerationContext gencont = factory.newWSSGenerationContext();
    // Initialize SAML issuer configuration via custom properties
    HashMap <Object, Object> customProps = new HashMap<Object,Object>();

    customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
    customProps.put(SamlConstants.TTL_PROP, "3600000");
    customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
    customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
    customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
    customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
    customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
    customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
    customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
    customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
    customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
    gencont.add(customProps); //Add custom properties
```

   Read about configuring a SAML token during token creation for more information about how you can specify configuration properties to control how the token is configured.

8. Add the Thin Client for JAX-WS JAR file to the classpath. Add *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the classpath. See the testing web services-enabled clients information for more information about adding this JAR file to the classpath.

9. Create the self-issued SAML token. The following code snippet illustrates creating the SAML sender-vouches token:

```
// Create SAMLToken
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");
System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the WSSFactory newSecurityToken method to specify how to create the SAML token.

      Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

      Creating a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the PolicyTool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken
```

      The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are creating. This object points to a SAMLGenerateCallbackHandler object that specifies the configuration parameters described in the following table:

Table 203. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Uses the constant value, `WSSConstants.SAML.SAML20_VALUE_TYPE` to specify a SAML 2.0 token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. In this scenario, specify the token value type by the valueType attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.SAML_NAME_IDENTIFIER	Specifies a user identity such as `myname` as the NameID value in the SAMLToken. If you do not define this parameter when using the Thin Client for JAX-WS, the NameID value does not contain useful information. If you are using a web services managed client, such a Java Platform, Enterprise Edition (Java EE) application making a web services request invocation, the Web Services Security runtime environment tries to extract user security information from the security context. Similarly, if you do not define this parameter for a managed web services client, the NameID value contains an UNAUTHENTICATED name identifier. This property is not used if your web services client has policy set attachments. Read about sending SAML tokens to learn more about sending the SAML token identity and attributes.	No
SamlConstants.SIGNATURE_REQUIRED	Specifies whether the issuer is required to digitally sign the SAML token. A `true` value specifies that issuer is required to digitally sign the SAML token. This value is the default.	No

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

Results

You have created a self-issued SAML token with the sender-vouches confirmation method with transport protection and then sent this token with web services request messages using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to create a self-issued SAML sender-vouches token and send that SAML token in web services request messages. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
  private String urlHost = "localhost";
  private String urlPort = "9081";
  private static final String CONTEXT_BASE = "/WSSampleSei/";
  private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
  private String message = "HELLO";
  private String uriString = "http://" + urlHost + ":" + urlPort;
  private String endpointURL = uriString + ECHO_CONTEXT12;
  private String input = message;

  /**
   * main()
   *
   * see printusage() for command-line arguments
   *
   * @param args
   */
  public static void main(String[] args) {
    SampleSamlSVClient sample = new SampleSamlSVClient();
    sample.CallService();
  }

  /**
   * CallService Parms were already read. Now call the service proxy classes.
   *
   */
  void CallService() {
    String response = "ERROR!:";
    try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf ");

      // Initialize WSSFactory object
      WSSFactory factory = WSSFactory.getInstance();
      // Initialize WSSGenerationContext
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      // Initialize SAML issuer configuration via custom properties
     HashMap<Object, Object> customProps = new HashMap<Object,Object>();

      customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
      customProps.put(SamlConstants.TTL_PROP, "3600000");
      customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
      customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
      customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
      customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
      customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
      customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
      customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
      gencont.add(customProps); //Add custom properties

      // Create SAMLToken
      HashMap<Object, Object> map = new HashMap<Object, Object>();
      map.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");
      map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
      map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
      map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
      SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
      SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml");

      System.out.println("SAMLToken id = " + samlToken.getId());

      // Initialize web services client
```

```
      EchoService12PortProxy echo = new EchoService12PortProxy();
      echo._getDescriptor().setEndpoint(endpointURL);

      // Configure SOAPAction properties
      BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
      Map<String, Object> requestContext = bp.getRequestContext();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
      requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
      requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

      gencont.add(samlToken);

      // Add timestamp
      WSSTimestamp timestamp = factory.newWSSTimestamp();
      gencont.add(timestamp);

      gencont.process(requestContext);

      // Build the input object
      EchoStringInput echoParm =
          new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
      echoParm.setEchoInput(input);
      System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

      // Prepare to consume timestamp in response message
      WSSConsumingContext concont = factory.newWSSConsumingContext();
      concont.add(WSSConsumingContext.TIMESTAMP);
      concont.process(requestContext);

      // Call the service
      response = echo.echoOperation(echoParm).getEchoResponse();

      System.out.println(">> CLIENT: SEI Echo invocation complete.");
      System.out.println(">> CLIENT: SEI Echo response is: " + response);
    } catch (Exception e) {
      System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
      e.printStackTrace();
    }
  }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Sending self-issued SAML holder-of-key tokens with symmetric key using WSS APIs:

You can create self-issued SAML tokens with the holder-of-key subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about sending self-issued SAML bearer tokens by using WSS APIs.
- Read about sending self-issued SAML sender-vouches tokens by using WSS APIs with message level protection.

About this task

This task focuses on using the symmetric key that is embedded in SAML security tokens to generate a digital signature of selected SOAP message elements in order to satisfy holder-of-key subject confirmation method security requirements. The Web Services Security policy attached to the web services provider is that of the *SAML20 HoK Symmetric WSSecurity default* policy set that is shipped in WebSphere Application Server 7.0.0.7 and later releases.

Procedure

1. Create a SAML security token that contains holder-of-key subject confirmation method; for example:

```
WSSFactory factory = WSSFactory.getInstance();
// Initialize WSSGenerationContext
com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext gencont = factory.newWSSGenerationContext();
// Initialize SAML issuer configuration via custom properties
HashMap<Object, Object> customProps = new HashMap<Object,Object>();

customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
customProps.put(SamlConstants.TTL_PROP, "3600000");
customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
gencont.add(customProps); //Add custom properties
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
map.put(SamlConstants.Token_REQUEST, "issue");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
map.put(SamlConstants.SERVICE_ALIAS, "soaprecipient");
map.put(SamlConstants.KEY_TYPE,
        "http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey");
map.put(SamlConstants.SAML_APPLIES_TO, "http://localhost:9080/your_Web_service");
map.put(RequesterConfiguration.RSTT.ENCRYPTIONALGORITHM,
        "http://www.w3.org/2001/04/xmlenc#aes256-cbc");
map.put(SamlConstants.KEY_SIZE, "256");
SAMLGenerateCallbackHandler callbackHandler = new
        SAMLGenerateCallbackHandler(map);
SAMLToken samlToken = (SAMLToken) factory.newSecurityToken(SAMLToken.class,
        callbackHandler, "system.wss.generate.saml");
```

   The embedded proof key in the SAML security token is encrypted for the target Web service. The public key of the target service that encrypts the proof key is specified by the `SamlConstants.SERVICE_ALIAS` property which specifies a public certificate in the trust file. The trust file location is specified by a `com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext` custom property. In this example, you must import the Java Cryptography Extension (JCE) policy file because encryption uses 256 bit key size. For more information, read about using the unrestricted JCE policy files in the "Tuning Web Services Security applications" topic.

   If you prefer to use derived keys for digital signing and for encryption instead of using symmetric key directly, add the following name-value pair:

```
map.put(SamlConstants.REQUIRE_DKT, "true");
```

2. Use the `WSSGenerationContext` object to prepare for request message security header processing; for example:

```
gencon.add(samlToken); //this line of code can be omitted

WSSTimestamp timestamp = factory.newWSSTimestamp();
gencon.add(timestamp);

WSSSignature sig = factory.newWSSSignature(samlToken);

sig.setSignatureMethod(WSSSignature.HMAC_SHA1);
sig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
sig.addSignPart(WSSSignature.BODY);
sig.addSignPart(WSSSignature.TIMESTAMP);
sig.addSignPart(WSSSignature.ADDRESSING_HEADERS);
sig.setTokenReference(SecurityToken.REF_KEYID);
//If the gencon.add(samlToken); line of code is omitted, or DerivedKey is used
//the above line of code must be replaced with
//sig.setTokenReference(SecurityToken.REF_STR);

gencon.add(sig);

WSSEncryption enc = factory.newWSSEncryption(samlToken);

enc.setEncryptionMethod(WSSEncryption.AES256);
enc.setTokenReference(SecurityToken.REF_KEYID);
//If the gencon.add(samlToken); line of code is omitted, or DerivedKey is used
//the above line of code must be replaced with
//enc.setTokenReference(SecurityToken.REF_STR);
```

```
enc.encryptKey(false);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);
enc.addEncryptPart(WSSEncryption.SIGNATURE);
gencon.add(enc);
```

3. Create the `WSSConsumingContext` object to prepare for response message, security header processing; for example:

```
WSSConsumingContext concont = factory.newWSSConsumingContext();

HashMap<Object, Object> map = new HashMap<Object, Object>();

SAMLConsumerCallbackHandler callbackHandler = new
        SAMLConsumerCallbackHandler(map);

WSSDecryption dec = factory.newWSSDecryption(SAMLToken.class, callbackHandler,
        "system.wss.consume.saml");
dec.addAllowedEncryptionMethod(WSSDecryption.AES256);
dec.encryptKey(false);
dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);

concont.add(dec);

callbackHandler = new SAMLConsumerCallbackHandler(map);
WSSVerification ver = factory.newWSSVerification(SAMLToken.class, callbackHandler,
        "system.wss.consume.saml");
ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);
ver.addRequiredVerifyPart(WSSVerification.BODY);
ver.addRequiredVerifyPart(WSSVerification.TIMESTAMP);

concont.add(ver);
```

4. Use the JDK `keytool` utility to generate the `saml-provider.jceks` and `recipient.jceks`files that are used to test the example code; for example:

```
keytool -genkey -alias samlissuer -keystore saml-provider.jceks -dname "CN=SAMLIssuer, O=ACME" -storepass issuerstorepass
 -keypass issuerkeypass -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -genkey -alias soaprecipient -keystore recipient.jceks -dname "CN=SOAPRecipient, O=ACME" -storepass reciptstorepass
 -keypass reciptkeypass -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -export -alias soaprecipient -file reciptpub.cer -keystore recipient.jceks -storepass reciptstorepass -storetype jceks

keytool -import -alias soaprecipient -file reciptpub.cer -keystore saml-provider.jceks -storepass issuerstorepass -storetype jceks
 -keypass issuerkeypass -noprompt
```

Results

You have learned key building blocks to create a web services client application to send a SAML security token in a SOAP message and to use the symmetric key that is embedded in SAML security in message level protection.

Sending self-issued SAML holder-of-key tokens with asymmetric key using WSS APIs:

You can create self-issued SAML tokens with the holder-of-key subject confirmation method and then use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to send these tokens with web services request messages.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about sending self-issued SAML bearer tokens by using WSS APIs.
- Read about sending self-issued SAML sender-vouches tokens by using WSS APIs with message level protection.

About this task

This task focuses on using the asymmetric key that is identified by SAML security tokens to generate a digital signature of selected SOAP message elements in order to satisfy holder-of-key subject confirmation method security requirements. The X.509 certificate of the sender is embedded in the SAML security token. The sender signs selected parts of request message elements by using its corresponding private key and encrypts the request message by using the public key of the recipient. The recipient signs the selected elements of the response message by using the private key of the recipient, and encrypts selected elements of the response message by using the public key of the sender in SAML security tokens. The Web services security policy attached to the web services provider is provided for your reference.

Procedure

1. Create a SAML security token that contains the holder-of-key subject confirmation method; for example:

```
WSSFactory factory = WSSFactory.getInstance();
// Initialize WSSGenerationContext
com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext gencont = factory.newWSSGenerationContext();
// Initialize SAML issuer configuration via custom properties
HashMap<Object, Object> customProps = new HashMap<Object,Object>();
customProps.put(SamlConstants.ISSUER_URI_PROP, "example.com");
customProps.put(SamlConstants.TTL_PROP, "3600000");
customProps.put(SamlConstants.KS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.KS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.KS_PW_PROP, "{xor}LCswLTovPiws");
customProps.put(SamlConstants.KEY_ALIAS_PROP, "samlissuer");
customProps.put(SamlConstants.KEY_NAME_PROP, "CN=SAMLIssuer, O=EXAMPLE");
customProps.put(SamlConstants.KEY_PW_PROP, "{xor}NDomLz4sLA==");
customProps.put(SamlConstants.TS_PATH_PROP, "keystores/saml-provider.jceks");
customProps.put(SamlConstants.TS_TYPE_PROP, "JCEKS");
customProps.put(SamlConstants.TS_PW_PROP, "{xor}LCswLTovPiws");
gencont.add(customProps); //Add custom properties
HashMap<Object, Object> map = new HashMap<Object, Object>();
map.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
map.put(SamlConstants.Token_REQUEST, "issue");
map.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
map.put(SamlConstants.SAML_NAME_IDENTIFIER, "Alice");
map.put(SamlConstants.SIGNATURE_REQUIRED, "true");
map.put(SamlConstants.KEY_TYPE,
        "http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey");
map.put(SamlConstants.SAML_APPLIES_TO, "http://localhost:9080/your_Web_service");
map.put(SamlConstants.KEY_ALIAS, "soapinitiator" );
map.put(SamlConstants.KEY_NAME, "CN=SOAPInitator, O=ACME");
map.put(SamlConstants.KEY_PASSWORD, "keypass");
map.put(SamlConstants.KEY_STORE_PATH, "keystores/initiator.jceks");
map.put(SamlConstants.KEY_STORE_PASSWORD, "storepass");
map.put(SamlConstants.KEY_STORE_TYPE, "jceks");
SAMLGenerateCallbackHandler callbackHandler = new SAMLGenerateCallbackHandler(map);
SAMLToken samlToken = (SAMLToken) factory.newSecurityToken(SAMLToken.class,
                      callbackHandler, "system.wss.generate.saml");
```

   The private key of the sender is specified by the `SamlConstants.KEY_ALIAS` property and is used to sign selected elements of the request message.

2. Use the `WSSGenerationContext` object to prepare for request message security header processing; for example:

```
gencon.add(samlToken); //this line of code can be omitted
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencon.add(timestamp);
WSSSignature sig = factory.newWSSSignature(samlToken);
sig.setTokenReference(SecurityToken.REF_KEYID);
//If the gencon.add(samlToken); line of code is omitted,
//the above line of code must be replaced with
//sig.setTokenReference(SecurityToken.REF_STR);

sig.setSignatureMethod(WSSSignature.RSA_SHA1);
sig.setCanonicalizationMethod(WSSSignature.EXC_C14N);
sig.addSignPart(WSSSignature.BODY);
sig.addSignPart(WSSSignature.TIMESTAMP);
sig.addSignPart(WSSSignature.ADDRESSING_HEADERS);
gencon.add(sig);
```

```
X509GenerateCallbackHandler x509callbackHandler2 = new X509GenerateCallbackHandler(
        null,
        "keystores/initiator.jceks",
        "jceks",
        "storepass".toCharArray(),
        "soaprecipient",
        null,
        "", null);
SecurityToken st2 = factory.newSecurityToken(X509Token.class, x509callbackHandler2);
WSSEncryption enc = factory.newWSSEncryption(st2);
enc.addEncryptPart(WSSEncryption.BODY_CONTENT);
enc.addEncryptPart(WSSEncryption.SIGNATURE);
enc.setEncryptionMethod(WSSEncryption.AES256);
enc.setKeyEncryptionMethod(WSSEncryption.KW_RSA_OAEP);
gencon.add(enc);
```

In this example, encryption uses a 256 bit key size so you must import the Java Cryptography Extension (JCE) policy file. For more information, read about using the unrestricted JCE policy files in the “Tuning Web Services Security applications” topic.

3. Create the `WSSConsumingContext` object to prepare for response message security header processing; for example:

```
WSSConsumingContext concont = factory.newWSSConsumingContext();

HashMap<Object, Object> map = new HashMap<Object, Object>();

SAMLConsumerCallbackHandler callbackHandler = new SAMLConsumerCallbackHandler(map);
WSSDecryption dec = factory.newWSSDecryption(SAMLToken.class, callbackHandler,
                            "system.wss.consume.saml");
dec.addAllowedEncryptionMethod(WSSDecryption.AES256);
dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_RSA_OAEP);
dec.encryptKey(false);
dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);
concont.add(dec);
X509ConsumeCallbackHandler verHandler = new X509ConsumeCallbackHandler(null,
            "keystores/initiator.jceks",
            "jceks",
            "storepass".toCharArray(),
            "soaprecipient",
            null, null);
WSSVerification ver = factory.newWSSVerification(X509Token.class, verHandler);
ver.addRequiredVerifyPart(WSSVerification.BODY);
concont.add(ver);
```

4. Use the JDK **keytool** utility to generate the `saml-provider.jceks`, `initiator.jceks`, and `recipient.jceks`files that are used to test the example code; for example:

```
keytool -genkey -alias samlissuer -keystore saml-provider.jceks -dname "CN=SAMLIssuer, O=ACME" -storepass storepass -keypass keypass
 -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -genkey -alias soaprecipient -keystore recipient.jceks -dname "CN=SOAPRecipient, O=ACME" -storepass storepass -keypass keypass
 -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -genkey -alias soapinitiator -keystore initiator.jceks -dname "CN=SOAPInitator, O=ACME" -storepass storepass -keypass keypass
 -storetype jceks -validity 5000 -keyalg RSA -keysize 2048

keytool -export -alias samlissuer -file issuerpub.cer -keystore saml-provider.jceks -storepass storepass -storetype jceks
keytool -export -alias soaprecipient -file reciptpub.cer -keystore recipient.jceks -storepass storepass -storetype jceks
keytool -export -alias soapinitiator -file initatpub.cer -keystore initiator.jceks -storepass storepass -storetype jceks

keytool -import -alias samlissuer -file issuerpub.cer -keystore initiator.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt
keytool -import -alias soaprecipient -file reciptpub.cer -keystore initiator.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt

keytool -import -alias samlissuer -file issuerpub.cer -keystore recipient.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt
keytool -import -alias soapinitiator -file initatpub.cer -keystore recipient.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt

keytool -import -alias soapinitiator -file initatpub.cer -keystore saml-provider.jceks -storepass storepass -storetype jceks -keypass keypass -noprompt
```

Results

You have learned key building blocks to create a web services client application to send a SAML security token in a SOAP message and to use the asymmetric key that is embedded in SAML security in message level protection.

Example

The following example illustrates the web services provider Web services security policy:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:spe="http://www.ibm.com/xmlns/prod/websphere/200605/ws-securitypolicy-ext">
    <wsp:Policy wsu:Id="response:app_encparts">
        <sp:EncryptedElements>
            <sp:XPath>/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Envelope']/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
             and local-name()='Security']/*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']</sp:XPath>
            <sp:XPath>/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
             and local-name()='Envelope']/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
             and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
             and local-name()='Security']/*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']</sp:XPath>
        </sp:EncryptedElements>
        <sp:EncryptedParts>
            <sp:Body/>
        </sp:EncryptedParts>
    </wsp:Policy>
    <wsp:Policy wsu:Id="request:req_enc">
        <sp:EncryptedParts>
            <sp:Body/>
        </sp:EncryptedParts>
    </wsp:Policy>
    <wsp:Policy wsu:Id="request:app_signparts">
        <sp:SignedParts>
            <sp:Body/>
            <sp:Header Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
            <sp:Header Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
        <sp:SignedElements>
            <sp:XPath>/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Envelope']/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
             and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
             and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'
             and local-name()='Timestamp']</sp:XPath>
            <sp:XPath>/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
            and local-name()='Envelope']/*[namespace-uri()='http://www.w3.org/2003/05/soap-envelope'
            and local-name()='Header']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd'
            and local-name()='Security']/*[namespace-uri()='http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd'
            and local-name()='Timestamp']</sp:XPath>
        </sp:SignedElements>
    </wsp:Policy>
    <wsp:Policy wsu:Id="response:resp_sig">
        <sp:SignedParts>
            <sp:Body/>
        </sp:SignedParts>
    </wsp:Policy>
    <sp:AsymmetricBinding>
        <wsp:Policy>
            <sp:InitiatorToken>
                <wsp:Policy>
                    <spe:CustomToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/Always"/>
                        <wsp:Policy>
                            <spe:WssCustomToken localname="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"/>
                        </wsp:Policy>
                    </spe:CustomToken>
                </wsp:Policy>
            </sp:InitiatorToken>
            <sp:AlgorithmSuite>
                <wsp:Policy>
                    <sp:Basic256/>
                </wsp:Policy>
            </sp:AlgorithmSuite>
            <sp:IncludeTimestamp/>
            <sp:RecipientToken>
                <wsp:Policy>
                    <sp:X509Token sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200512/IncludeToken/Always"/>
                        <wsp:Policy>
                            <sp:WssX509V3Token11/>
                        </wsp:Policy>
                    </sp:X509Token>
                <wsp:Policy>
            <sp:RecipientToken>
            <sp:Layout>
                <wsp:Policy>
                    <sp:Strict/>
                </wsp:Policy>
            </sp:Layout>
        <wsp:Policy>
    <sp:AsymmetricBinding>
</wsp:Policy>
```

Requesting SAML bearer tokens from an external STS using WSS APIs and transport level protection:

You can request SAML tokens with the bearer subject confirmation method from an external Security Token Service (STS). After obtaining the SAML bearer token, you can then send these tokens with web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings.

About this task

You can request a SAML token with the bearer subject confirmation method from an external STS and then send the SAML token in web services request messages from a web services client using WSS APIs.

The web services application client used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code snippets from the sample are described in the procedure section, and a complete, ready-to-use web services client sample is provided in the Example section.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

      The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

   Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS web services thin client; however, you can also use a managed client.

2. Attach the SAML20 Bearer WSHTTPS default policy set to the web services provider. This policy set is used to protect messages using HTTPS transport. Read about configuring client and provider bindings for the SAML Bearer token for details on how to attach the SAML20 Bearer WSHTTPS default policy set to the web services provider.

3. Assign the SAML Bearer Provider sample default general bindings to the sample web services provider. Read about configuring client and provider bindings for the SAML bearer token for details on assigning the SAML Bearer Provider sample default general bindings to your web services application.

4. Verify that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate. Using the administrative console, complete the following steps:
   a. Click **Services** > **Policy sets** > **General provider policy set bindings** > **Saml Bearer Provider sample** > **WS-Security** > **Authentication and protection**.
   b. Click **gen_saml11token** in the Authentication tokens table.
   c. Click **Callback handler**.
   d. In the Custom Properties section, ensure that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate.
5. Request the SAML token from an external STS. The following code snippet illustrates how to request the SAML token and assumes that an external STS is configured to accept a UsernameToken, and to issue a SAML 2.0 token after validation:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.
   b. Use the WSSFactory newSecurityToken method to request a SAML token from an external STS.

      Specify the following method to request the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

      Requesting a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken"
```

      The `SAMLToken.class` parameter specifies the type of security token to create.

      The `callbackHandler` object contains parameters that define the characteristics of the SAML token that you are requesting and other parameters required to reach the STS and obtain the SAML token. The SAMLGenerateCallbackHandler object specifies the configuration parameters described in the following table:

Table 204. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the bearer subject confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `Bearer` confirmation method.	Yes

Table 204. SAMLGenerateCallbackHandler properties (continued). This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the bearer subject confirmation method.

Property	Description	Required
SamlConstants.TOKEN_TYPE	Specifies the token type. When a web services client has policy set attachments, this property is not used by Web Services Security runtime environment. Specify the token value type by using the `valueType` attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 2.0 token; however, you can also use the `WSSConstants.SAML.SAML11_VALUE_TYPE` value.	Yes
SamlConstants.STS_ADDRESS	Specifies the Security Token Service address. For the example used in this task topic, the value of this property is set to `https` to specify to use SSL to protect the SAML Token request. You must set the -Dcom.ibm.SSL.ConfigURL property to enable the use of SSL to protect the SAML token request with the STS.	Yes
SamlConstants.SAML_APPLIES_TO	Specifies the target STS address for where you want to use the SAML token.	No
SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request from the STS a single token that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a `true` value for this property indicates to request multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS.	No
SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request.	No

A WSSGenerationContext instance and a WSSConsumingContext instance are also set in the SAMLGenerateCallbackHandler object. The WSSGenerationContext instance must contain a UNTGenerateCallbackHandler object with the information to create the UsernameToken that you want to send to the STS.

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

c. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

6. Add the SAML token to the SOAP security header of a web services request messages.

a. Initialize the web services client and configure the SOAPAction properties. The following code snippet illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);
```

```
// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");
```

b. Initialize the WSSGenerationContext. The following code illustrates the use of the WSSGenerationContext interface to initialize a generation context and enable you to insert the `SAMLToken` into the web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

c. Add the timestamp element in the SOAP messages security header. The SAML20 Bearer WSHTTPS default policy set requires web services requests and response messages to carry a timestamp element in SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call specifies the timestamp to put into a request message:

```
// Add a timestamp to the request message.
WSSTimestamp timestamp = factory.newWSSTimestamp();
gencont.add(timestamp);

gencont.process(requestContext);
```

d. Attach WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

e. Specify SSL transport level message protection using JVM properties.

The SAML20 Bearer WSHTTPS default policy set requires transport-level message protection using SSL. In addition, you can use this same property to enable protection of the SAML token request to the STS using SSL. Specify SSL transport-level message protection using the following JVM property:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

Results

You have requested a SAML token with the bearer subject confirmation method with transport level protection from an external STS. After obtaining the token, you sent the token with web services request messages using the JAX-WS programming model and WSS APIs.

If you want to request a SAML token with the bearer subject confirmation method with message level protection from an external STS, see the documentation for requesting SAML sender-vouches tokens from an external STS using WSS APIs and message level protection. To use message level protection for SAML tokens with the bearer subject confirmation method, in the step to request the SAML token from an external STS, specify a confirmation method of `Bearer` instead of `sender-vouches`; for example:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
```

```
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

Additionally, the step to configure the verification of the digital signature in the response message is optional in the case of the bearer token.

Example

The following code sample is a web services client application that demonstrates how to request a SAML token from an external STS and send that SAML token in web services request messages. If your usage scenario requires SAML tokens, but does not require your application to pass the SAML tokens using web services messages, you only need to use the first part of the following sample code, up through the `// Initialize web services client` section.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */

package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

/**
 * SampleClient
 * main entry point for thin client JAR sample
 * and worker class to communicate with the services
 */
public class SampleClient {

 private String urlHost = "localhost";
 private String urlPort = "9443";
 private static final String CONTEXT_BASE = "/WSSampleSei/";
 private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
 private String message = "HELLO";
 private String uriString = "https://" + urlHost + ":" + urlPort;
 private String endpointURL = uriString + ECHO_CONTEXT12;
 private String input = message;

 /**
  * main()
  *
  * see printusage() for command-line arguments
  *
  * @param args
  */
```

```
 public static void main(String[] args) {
  SampleClient sample = new SampleClient();
  sample.CallService();
 }

 /**
  * CallService Parms were already read. Now call the service proxy classes
  *
  */
 void CallService() {
  String response = "ERROR!:";
  try {
      System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
   System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");

//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML20_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "Bearer");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);


// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());


        // Initialize web services client
   EchoService12PortProxy echo = new EchoService12PortProxy();
   echo._getDescriptor().setEndpoint(endpointURL);

   // Configure SOAPAction properties
   BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
   Map<String, Object> requestContext = bp.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
   requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

   // Initialize WSSGenerationContext
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
        gencont.add(samlToken);

        // Add timestamp
        WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);

        gencont.process(requestContext);

        // Build the input object
   EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
   echoParm.setEchoInput(input);
   System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

   // Prepare to consume timestamp in response message.
   WSSConsumingContext concont = factory.newWSSConsumingContext();
       concont.add(WSSConsumingContext.TIMESTAMP);
       concont.process(requestContext);

   // Call the service
   response = echo.echoOperation(echoParm).getEchoResponse();

   System.out.println(">> CLIENT: SEI Echo invocation complete.");
   System.out.println(">> CLIENT: SEI Echo response is: " + response);
  } catch (Exception e) {
   System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
   e.printStackTrace();
  }
 }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _191EBC44865015D9AB1270745072344
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to https://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12===>>HELLO
```

Requesting SAML sender-vouches tokens from an external STS using WSS APIs and message level protection:

You can request SAML tokens with the sender-vouches subject confirmation method from an external Security Token Service (STS). After obtaining the SAML sender-vouches token, you can then send these tokens with web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API) with message level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, SSL transport protection, X.509 security token, and the use of policy sets to configure and administer web services settings.

About this task

You can request a SAML token with the sender-vouches subject confirmation method from an external STS and then send the SAML token in web services request messages from a web services client using WSS APIs with message level protection.

This product does not provide a default policy set that requires SAML tokens with sender-vouches subject confirmation method. Read about configuring client and provider bindings for the SAML sender-vouches token to learn more about how to create a Web Services Security policy to require SAML tokens with sender-vouches subject confirmation and how to create a custom binding configuration. You must attach the policy and binding to the web services provider. The code sample described in this task assumes that the web services provider policy requires that both the SAML tokens and the message bodies are digitally signed by using an X.509 security token.

The web services client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure, and a complete, ready-to-use web services client sample is provided.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider.

   Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

   The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

   To obtain and modify the sample web services client to add the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code.

      For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

The JaxWSServicesSamples.ear enterprise application and supporting Java archives (JAR) files are located in the installableApps directory within the JaxWSServicesSamples sample application.

c. Deploy the JaxWSServicesSamples.ear file onto the application server. After you deploy the JaxWSServicesSamples.ear file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS web services thin client; however, you can also use a managed client.

2. Specify to use SSL message-level message protection. Use the following JVM property to specify to use SSL to protect the SAML token request with the STS:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

3. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.

4. Request the SAML token from an external STS. The following code snippet illustrates how to request the SAML sender-vouches token and assumes that an external STS is configured to accept a Username token, and to issue a SAML 2.0 token using sender-vouches after validation:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

a. Use the WSSFactory newSecurityToken method to specify how to request the SAML token from an external STS.

Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

Requesting a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken"
```

The SAMLToken.class parameter specifies the type of security token to create.

The callbackHandler object contains parameters that define the characteristics of the SAMLToken that you are requesting and other parameters required to reach the STS and obtain the SAML token. The SAMLGenerateCallbackHandler object specifies the configuration parameters described in the following table:

Table 205. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Specifies the token type. When a web services client has policy set attachments, this property is not used by the Web Services Security runtime environment. Specify the token value type by using the `valueType` attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 1.1 token; however, you can also use the `WSSConstants.SAML.SAML20_VALUE_TYPE` value.	Yes
SamlConstants.STS_ADDRESS	Specifies the Security Token Service address. For the example used in this task topic, the value of this property is set to `https` to specify to use SSL to protect the SAML Token request. You must set the -Dcom.ibm.SSL.ConfigURL property to enable the use of SSL to protect the SAML token request with the STS.	Yes
SamlConstants.SAML_APPLIES_TO	Specifies the target STS address for where you want to use the SAML token.	No
SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request from the STS a single token that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a `true` value for this property indicates to request multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS.	No
SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request. The default value is WSTrust 1.3.	No

A WSSGenerationContext instance and a WSSConsumingContext instance are also set in the SAMLGenerateCallbackHandler object. The WSSGenerationContext instance must contain a UNTGenerateCallbackHandler object with the information to create the UsernameToken that you want to send to the STS.

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

5. Add the SAML token to the SOAP security header of web services request messages.

a. Initialize the web services client and configure the SOAPAction properties. The following code example illustrates these actions:

```
// Initialize web services client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

b. Initialize the WSSGenerationContext. The following code snippet illustrates the use of the gencont.object of the WSSGenerationContext type to initialize a generation context to enable you to insert the `SAMLToken` into a web services request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(samlToken);
```

Specifically, the `gencont.add(samlToken)` method call specifies to put the SAML token into a request message. This operation requires the client code to have the following Java 2 Security permission:

```
permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

6. Add an X.509 token for message protection using the Web Services Security API.

   This sample code uses the dsig-sender.ks key file and the SOAPRequester sample key. You must not use the sample key in a production environment. The following code snippet illustrates adding an X.509 token for message protection:

```
// Add an X.509 Token for message protection
    X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
         null,
         "profile_root/etc/ws-security/samples/dsig-sender.ks",
         "JKS",
         "client".toCharArray(),
         "soaprequester",
         "client".toCharArray(),
         "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

   SecurityToken x509 = factory.newSecurityToken(X509Token.class,
   x509callbackHandler, "system.wss.generate.x509");

   WSSSignature sig = factory.newWSSSignature(x509);
   sig.setSignatureMethod(WSSSignature.RSA_SHA1);

   WSSSignPart sigPart = factory.newWSSSignPart();
   sigPart.setSignPart(samlToken);
   sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
   sig.addSignPart(sigPart);
   sig.addSignPart(WSSSignature.BODY);
```

a. Create a WSSSignature object with the X509 token. The following line of code creates a WSSSignature object with the X509 token:

```
WSSSignature sig = factory.newWSSSignature(x509);
```

b. Add the signed part to use for message protection. The following line of code specifies to add WSSSignature.BODY as the signed part:

```
sig.addSignPart(WSSSignature.BODY);
```

c. Add the timestamp element in the SOAP messages security header. The SAML20 SenderVouches WSHTTPS and SAML11 SenderVouches WSHTTPS policy sets require web services requests and response messages to carry a timestamp element in the SOAP messages Security header. In the following code snippet, the `factory.newWSSTimestamp()` method call generates the timestamp, and the `gencont.add(timestamp)` method call adds the timestamp into the request message:

```
// Add Timestamp
WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);


WSSConsumingContext concont = factory.newWSSConsumingContext();
```

d. Configure the SAML token signature using STR-Transform transform algorithm.

   A separate WSSSignPart is needed to specify the SecurityTokenReference transformation algorithm that is represented by the `WSSSignPart.TRANSFORM_STRT10` attribute. A SAML Token cannot be digitally signed directly. This attribute enables the Web Services Security runtime environment to generate a SecurityTokenReference element to reference the SAMLToken and to digitally sign the SAMLToken using the SecurityTokenReference transformation. The following line of code specifies to use the `WSSSignPart.TRANSFORM_STRT10` attribute:

```
WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(samlToken);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
```

e. Attach the WSSGenerationContext object to the web services RequestContext object. The WSSGenerationContext object now contains all the security information that is required to format a request message. The `gencont.process(requestContext)` method call attaches the WSSGenerationContext object to the web services RequestContext object to enable the Web Services Security runtime environment to format the required SOAP security header; for example:

```
// Attaches the WSSGenerationContext object to the web services RequestContext object.
gencont.process(requestContext);
```

7. Use the X.509 token to validate the digital signature and the integrity of the response message. If the provider policy requires the response message to be digitally signed, you must initialize the X.509 token.

   a. A X509ConsumeCallbackHandler object is initialized with a truststore, dsig-receiver.ks, and a certificate path object to validate the provider digital signature. The following line of code is used to initialize the X509ConsumeCallbackHandler object:

```
X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
 "profile_root/etc/ws-security/samples/dsig-receiver.ks",
          "JKS",
           "server".toCharArray(),
            certList,
                 java.security.Security.getProvider("IBMCertPath"));
```

   b. A WSSVerification object is created and the message body is added to the verification object so that the Web Services Security runtime environment validates the digital signature.

      The following line of code is used to initialize the WSSVerification object:

```
WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);
```

      The WSSConsumingContext object now contains all the security information that is required to format a request message. The `concont.process(requestContext)` method call attaches the WSSConsumingContext object to the response method; for example:

```
// Attaches the WSSConsumingContext object to the web services RequestContext object.
concont.process(requestContext);
```

Results

You have requested a SAML token with the sender-vouches confirmation method from an external STS. After obtaining the token, you sent the token with web services request messages using message level protection using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to request a SAML token from an external STS and send that SAML token in web services request messages with message level protection. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
```

```
import com.ibm.was.wssample.sei.echo.EchoStringInput;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSException;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignPart;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignature;
import com.ibm.websphere.wssecurity.wssapi.verification.WSSVerification;
import com.ibm.websphere.wssecurity.wssapi.token.X509Token;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.security.InvalidAlgorithmParameterException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.cert.CertStore;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.CollectionCertStoreParameters;
import java.security.cert.X509Certificate;
import java.util.HashSet;
import java.util.Set;
import java.util.HashMap;
import java.util.Map;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
 private String urlHost = "localhost";
 private String urlPort = "9080";
 private static final String CONTEXT_BASE = "/WSSampleSei/";
 private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
 private String message = "HELLO";
 private String uriString = "http://" + urlHost + ":" + urlPort;
 private String endpointURL = uriString + ECHO_CONTEXT12;
 private String input = message;

 /**
  * main()
  *
  * see printusage() for command-line arguments
  *
  * @param args
  */
 public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
 }

 /**
  * CallService Parms were already read. Now call the service proxy classes.
  *
  */
 void CallService() {
  String response = "ERROR!:";
  try {
       System.setProperty("com.ibm.SSL.ConfigURL", "profile_root/properties/ssl.client.props");
       System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas.conf");


//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
```

```
gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());

         // Initialize web services client.
   EchoService12PortProxy echo = new EchoService12PortProxy();
   echo._getDescriptor().setEndpoint(endpointURL);

   // Configure SOAPAction properties
   BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
   Map<String, Object> requestContext = bp.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
   requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

   // Initialize WSSGenerationContext
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
         gencont.add(samlToken);

         // Add X.509 Tokens for message protection
            X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
           null,
           "profile_root/etc/ws-security/samples/dsig-sender.ks",
           "JKS",
           "client".toCharArray(),
           "soaprequester",
           "client".toCharArray(),
           "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

            SecurityToken x509 = factory.newSecurityToken(X509Token.class,
            x509callbackHandler, "system.wss.generate.x509");

            WSSSignature sig = factory.newWSSSignature(x509);
            sig.setSignatureMethod(WSSSignature.RSA_SHA1);

            WSSSignPart sigPart = factory.newWSSSignPart();
            sigPart.setSignPart(samlToken);
            sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
            sig.addSignPart(sigPart);
            sig.addSignPart(WSSSignature.BODY);

         // Add timestamp
         WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);
         sig.addSignPart(WSSSignature.TIMESTAMP);

         gencont.add(sig);


         WSSConsumingContext concont = factory.newWSSConsumingContext();

         // Prepare to consume timestamp in response message
       concont.add(WSSConsumingContext.TIMESTAMP);

       // Prepare to verify digital signature in response message
         X509Certificate x509cert = null;
         try {
                 InputStream is = new FileInputStream("profile_root/etc/ws-security/samples/intca2.cer");
                 CertificateFactory cf = CertificateFactory.getInstance("X.509");
                 x509cert = (X509Certificate) cf.generateCertificate(is);
         } catch (FileNotFoundException e1) {
                 throw new WSSException(e1);
         } catch (CertificateException e2) {
                 throw new WSSException(e2);
         }
         Set<Object> eeCerts = new HashSet<Object>();
         eeCerts.add(x509cert);

         java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
         CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);

         CertStore cert = null;
         try {
                 cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
         } catch (NoSuchProviderException e1) {
                 throw new WSSException(e1);
         } catch (InvalidAlgorithmParameterException e2) {
                 throw new WSSException(e2);
         } catch (NoSuchAlgorithmException e3) {
                 throw new WSSException(e3);
         }
         if (certList != null) {
                 certList.add(cert);
         }
```

```
        X509ConsumeCallbackHandler callbackHandlerVer = new X509ConsumeCallbackHandler(
          "profile_root/etc/ws-security/samples/dsig-receiver.ks",
          "JKS",
          "server".toCharArray(),
          certList,
          java.security.Security.getProvider("IBMCertPath"));

        WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);

        ver.addRequiredVerifyPart(WSSVerification.BODY);
        concont.add(ver);

        gencont.process(requestContext);
        concont.process(requestContext);

        // Build the input object
   EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
   echoParm.setEchoInput(input);
   System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

   // Call the service
   response = echo.echoOperation(echoParm).getEchoResponse();

   System.out.println(">> CLIENT: SEI Echo invocation complete.");
   System.out.println(">> CLIENT: SEI Echo response is: " + response);
  } catch (Exception e) {
   System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
   e.printStackTrace();
  }
 }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12==>>HELLO
```

Requesting SAML sender-vouches tokens from an external STS using WSS APIs and transport level protection:

You can request SAML tokens with the sender-vouches subject confirmation method from an external Security Token Service (STS). After obtaining the SAML sender-vouches token, you can then send these tokens with web services request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API) with transport level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, SSL transport protection, and the use of policy sets to configure and administer web services settings.

About this task

You can request a SAML token with the sender-vouches subject confirmation method from an external STS and then send the SAML token in web services request messages from a web services client using WSS APIs with transport level protection.

The web services client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure, and a complete, ready-to-use web services client sample is provided.

Procedure

1. Identify and obtain the web services client that you want to use to invoke a web services provider. Use this client to insert SAML tokens in SOAP request messages programmatically using WSS APIs.

The web services client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web services sample application.

To obtain and modify the sample web services client to add the Web Services Security API to pass SAML sender-vouches tokens in SOAP request messages programmatically using WSS APIs, complete the following steps:

a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

b. Obtain the JaxWSServicesSamples client code.

   For example purposes, this procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web services Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

   The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web services client code against the sample application.

Instead of using the web services client sample, you can choose to add the code snippets to pass SAML tokens in SOAP request messages programmatically using WSS APIs in your own web services client application. The example in this procedure uses a JAX-WS Web services thin client; however, you can also use a managed client.

2. Create a copy of either the SAML20 Bearer WSHTTPS default policy set or the SAML11 Bearer WSHTTPS default policy set.

   Provide a name for the copy of the policy set; for example SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS to help you identify that this new policy set uses the sender-vouches confirmation method.

   No additional change is required to the new policy file because the subject confirmation method is specified in the binding configuration and not in the policy.

   The new policy file contains either SAMLToken20Bearer or the SAMLToken11Bearer as the policy identifiers. Change the identifier of the SAMLToken20Bearer policy to SAMLToken20SV or change the identifier of the SAMLToken11Bearer policy to SAMLToken11SV to specify a more descriptive name. Changing the identifier of the policy does not change the policy enforcement in any way; however, adding a descriptive identifier helps you to identify that these policy identifiers use the sender-vouches confirmation method.

   If you want to view the settings of these policies, use the administrative console to complete the following actions:

   a. Click **Services** > **Policy sets** > **Application policy sets** > ***policy_set_name***.

   b. Click the **WS-Security** policy in the policies table.

   c. Click the **Main policy** link or the **Bootstrap policy** link.

   d. Click **Request token policies** from the Policy Details section.

3. Attach the new SAML20 SenderVouches WSHTTPS or SAML11 SenderVouches WSHTTPS policy set to the web services provider application. Read about configuring client and provider bindings for the SAML sender-vouches token for details on attaching this policy set to your web services provider application.

4. Create a copy of the SAML Bearer Provider sample default general bindings.

   a. For the new copy of the default policy set, provide a name that includes sender-vouches, such as `SAML Sender-vouches provider binding`.

   b. In the callback handler of your SAML11 or SAML20 token consumer, change the value of the confirmationMethod property to `sender-vouches` in the token consumer configuration for the intended SAML token version. Ensure that the custom properties trustStoreType, trustStorePassword and trustStorePath correspond to the trust store containing the STS signer

certificate. Read about configuring client and provider bindings for the SAML sender-vouches token for details on modifying the sender-vouches bindings to satisfy the vouching requirement.

5. Assign the new provider binding to the JaxWSServicesSamples provider sample. Read about configuring client and provider bindings for the SAML sender-vouches for details on assigning the SAML sender-vouches provider sample, default general bindings to your web services provider application.
6. Enable the web services provider SSL configuration attribute, clientAuthentication, to require X.509 client certificate authentication.

   The clientAuthentication attribute determines whether SSL client authentication is required. To specify the clientAuthentication attribute, use the administrative console to complete the following actions:

   a. Click **Security** > **SSL certificates and key management** > **Manage endpoint security configurations** > **{Inbound | Outbound}** > ***SSL_configuration***.
   b. Click the **WC_defaulthost_secure** link in the inbound topology.
   c. From Related Items, click the **SSL_configurations** link.
   d. Select the **NodeDefaultSSLSettings** resource.
   e. Click **Quality of protection (QoP) settings** link.
   f. Select **Required** from the menu to specify client authentication.

   Read about creating a secure sockets layer configuration to learn more about configuring the clientAuthentication attribute.
7. Specify to use SSL transport-level message protection. Use the following JVM property to specify to use SSL to protect the SAML token request with the STS:

```
-Dcom.ibm.SSL.ConfigURL=file:profile_root\properties\ssl.client.props
```

   Alternatively, you can define the SSL configuration file using a Java system property in the sample client code; for example:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

8. Add the Thin Client for JAX-WS JAR file to the class path. Add the *app_server_root*/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar file to the class path. See the testing web services-enabled clients information for more information about adding this JAR file to the class path.
9. Request the SAML token from an external STS. The following code snippet illustrates how to request the SAML sender-vouches token and assumes that an external STS is configured to accept a UsernameToken, and to issue a SAML 1.1 token using sender-vouches after validation:

```
//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());
```

   a. Use the WSSFactory newSecurityToken method to specify how to request the SAML token from an external STS.

      Specify the following method to create the SAML token:

```
WSSFactory  newSecurityToken(SAMLToken.class, callbackHandler, "system.wss.generate.saml")
```

Requesting a SAML token requires the Java security permission wssapi.SAMLTokenFactory.newSAMLToken. Use the Policy Tool to add the following policy statement to the Java security policy file or the application client was.policy file:

```
permission java.security.SecurityPermission "wssapi.SAMLTokenFactory.newSAMLToken"
```

The `SAMLToken.class` parameter specifies the type of security token to create.

The `callbackHandler` object contains parameters that define the characteristics of the SAMLToken that you are requesting and other parameters required to reach the STS and obtain the SAML token. The SAMLGenerateCallbackHandler object specifies the configuration parameters described in the following table:

Table 206. SAMLGenerateCallbackHandler properties. This table describes the configuration parameters for the SAMLGenerateCallbackHandler object using the sender-vouches confirmation method.

Property	Description	Required
SamlConstants.CONFIRMATION_METHOD	Specifies to use the `sender-vouches` confirmation method.	Yes
SamlConstants.TOKEN_TYPE	Specifies the token type. When a web services client has policy set attachments, this property is not used by the Web Services Security runtime environment. Specify the token value type by using the `valueType` attribute of the tokenGenerator binding configuration. The example in this procedure uses a SAML 1.1 token; however, you can also use the `WSSConstants.SAML.SAML20_VALUE_TYPE` value.	Yes
SamlConstants.STS_ADDRESS	Specifies the Security Token Service address. For the example used in this task topic, the value of this property is set to `https` to specify to use SSL to protect the SAML Token request. You must set the -Dcom.ibm.SSL.ConfigURL property to enable the use of SSL to protect the SAML token request with the STS.	Yes
SamlConstants.SAML_APPLIES_TO	Specifies the target STS address for where you want to use the SAML token.	No
SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST	Specifies whether to request from the STS a single token that is enclosed in a RequestSecurityToken (RST) element or multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element. The default behavior is to request a single token that is enclosed in a RequestSecurityToken (RST) element from the STS. Specifying a `true` value for this property indicates to request multiple tokens in a collection of RST elements that are enclosed in a single RequestSecurityTokenCollection (RSTC) element from the STS.	No
SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE	Specifies the WS-Trust namespace that is included in the WS-Trust request. The default value is WSTrust 1.3.	No

A WSSGenerationContext instance and a WSSConsumingContext instance are also set in the SAMLGenerateCallbackHandler object. The WSSGenerationContext instance must contain a UNTGenerateCallbackHandler object with the information to create the UsernameToken that you want to send to the STS.

The `system.wss.generate.saml` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the SAML token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

b. Obtain the token identifier of the created SAML token.

Use the following statement as a simple test for the SAML token that you created:

```
System.out.println("SAMLToken id = " + samlToken.getId())
```

Results

You have requested a SAML token with the sender-vouches confirmation method from an external STS. After obtaining the token, you sent the token with web services request messages using transport protection using the JAX-WS programming model and WSS APIs.

Example

The following code sample is a complete, ready-to-use web services client application that demonstrates how to request a SAML token from an external STS and send that SAML token in web services request messages with transport level protection. This sample code illustrates the procedure steps described previously.

```
/**
 * The following source code is sample code created by IBM Corporation.
 * This sample code is provided to you solely for the purpose of assisting you in the
 * use of the technology.  The code is provided 'AS IS', without warranty or condition of
 * any kind.  IBM shall not be liable for any damages arising out of your use of the
 * sample code, even if IBM has been advised of the possibility of such damages.
 */
package com.ibm.was.wssample.sei.cli;

import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;

import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.callbackhandler.SAMLGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.wssapi.token.SAMLToken;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.saml.config.SamlConstants;

import java.util.Map;
import java.util.HashMap;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
 private String urlHost = "localhost";
 private String urlPort = "9443";
 private static final String CONTEXT_BASE = "/WSSampleSei/";
 private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
 private String message = "HELLO";
 private String uriString = "https://" + urlHost + ":" + urlPort;
 private String endpointURL = uriString + ECHO_CONTEXT12;
 private String input = message;

 /**
  * main()
  *
  * see printusage() for command-line arguments
  *
  * @param args
  */
 public static void main(String[] args) {
  SampleSamlSVClient sample = new SampleSamlSVClient();
  sample.CallService();
 }

 /**
  * CallService Parms were already read. Now call the service proxy classes.
  *
  */
 void CallService() {
  String response = "ERROR!:";
  try {

         System.setProperty("com.ibm.SSL.ConfigURL", "profile_root//properties/ssl.client.props");
         System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");


//Request the SAML Token from external STS
WSSFactory factory = WSSFactory.getInstance();
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
```

```
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, ENDPOINT_URL);
cbackMap1.put(SamlConstants.TRUST_CLIENT_WSTRUST_NAMESPACE, "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(SamlConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
cbackMap1.put(SamlConstants.TOKEN_TYPE, WSSConstants.SAML.SAML11_VALUE_TYPE);
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "sender-vouches");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);

// Add UNT to trust request
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);

gencont1.add(ut);

cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class, cbHandler1, "system.wss.generate.saml");

System.out.println("SAMLToken id = " + samlToken.getId());


        // Initialize web services client
   EchoService12PortProxy echo = new EchoService12PortProxy();
   echo._getDescriptor().setEndpoint(endpointURL);

   // Configure SOAPAction properties
   BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
   Map<String, Object> requestContext = bp.getRequestContext();
   requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
   requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
   requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

   // Initialize WSSGenerationContext
   WSSGenerationContext gencont = factory.newWSSGenerationContext();
         gencont.add(samlToken);


         // Add timestamp
         WSSTimestamp timestamp = factory.newWSSTimestamp();
   gencont.add(timestamp);

         gencont.process(requestContext);

         // Build the input object
   EchoStringInput echoParm =
    new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
   echoParm.setEchoInput(input);
   System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

   // Prepare to consume timestamp in response message
   WSSConsumingContext concont = factory.newWSSConsumingContext();
       concont.add(WSSConsumingContext.TIMESTAMP);
       concont.process(requestContext);

   // Call the service
   response = echo.echoOperation(echoParm).getEchoResponse();

   System.out.println(">> CLIENT: SEI Echo invocation complete.");
   System.out.println(">> CLIENT: SEI Echo response is: " + response);
  } catch (Exception e) {
   System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
   e.printStackTrace();
     }
 }
}
```

When this web services client application sample runs correctly, you receive messages like the following messages:

```
SAMLToken id = _6CDDF0DBF91C044D211271166233407
Retrieving document at 'file:profile_root/.../wsdl/'.
>> CLIENT: SEI Echo to http://localhost:9443/WSSampleSei/EchoService12
>> CLIENT: SEI Echo invocation complete.
>> CLIENT: SEI Echo response is: SOAP12===>>HELLO
```

Requesting SAML holder-of-key tokens with symmetric key from external security token service using WSS APIs:

You can request an external security token service (STS) to issue SAML tokens with the holder-of-key subject confirmation method with symmetric key that is encrypted for a target service. Use the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs) to complete this task.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about propagating self-issued SAML holder-of-key tokens with symmetric key by using WSS APIs.
- Become familiar with using embedded key materials in SAML tokens for message protection by using WSS APIs. Your usage scenario requires requesting SAML tokens from an external STS instead of using self-issued SAML tokens.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with message level protection.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with transport level protection.
- Read about requesting SAML bearer tokens from an external STS to propagate by using WSS APIs with transport level protection.
- Be familiar with accessing an external STS by using WSS APIs.

About this task

This task shows example code to request SAML tokens from an external STS, with holder-of-key subject confirmation method and embedded symmetric key that is encrypted for the target service by using WSS APIs. This task focuses on sending a WS-Trust request message to an external STS to request SAML holder-of-key tokens with symmetric keys.

Procedure

1. Specify an STS from which to request a SAML security token that contains holder-of-key subject confirmation method; for example:

```
com.ibm.websphere.wssecurity.wssapi.WSSFactory factory =
    com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, "https://www.example.com/sts");  //STS URL
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, "http://myhost:9080/myService");  //Target Service
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_SOAP_VERSION, "1.1");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
                  "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST,
              "true");   //RST or RSTC
cbackMap1.put(SamlConstants.TOKEN_TYPE,
              "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0");
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
```

   To request a holder-of-key SAML security token from the STS, you must specify whether to embed a symmetric key or a public key by way of a `KeyType` element in a trust request. This example requires a symmetric key type as shown in the next step.

2. Specify the symmetric key to be embedded in SAML security tokens; for example:

```
cbackMap1.put(SamlConstants.KEY_TYPE,
             "http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class,
                                                   cbHandler1, "system.wss.generate.saml");
```

The requested SAML token contains a symmetric key that is encrypted for the target service. The STS also returns the unencrypted symmetric key through the `WS-Trust RequestedProofToken` element. See the following example.

```
<wst:RequestedProofToken>
    <wst:BinarySecret
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
      wsu:Id="_27325D34CE4BCC83141288966548620">n68rFQba+XTZLNBfEc4prg==</wst:BinarySecret>
</wst:RequestedProofToken>
```

The `RequestedProofToken` element is shown here for your information. The detailed processing is not exposed to WSS APIs users. The `RequestedProofToken` element and the symmetric key are handled by the Web Services Security runtime environment, or more precisely by the `SAMLGenerateLoginModule` that is specified in the `system.wss.geenrate.saml` JAAS login configuration.

Results

You have learned key building blocks for requesting SAML tokens with holder-of-key subject confirmation method and symmetric key from an external STS by using WSS APIs. To use the SAML token to sign request messages, review the example code in the “Propagating self-issued SAML holder-of-key tokens with symmetric key by using WSS APIs” topic.

Requesting SAML holder-of-key tokens with asymmetric key from External Security Token Service using WSS APIs:

You can request an external Security Token Service (STS) to issue SAML tokens with the holder-of-key subject confirmation method with a public key in an X.509 certificate with the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS APIs).

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, SAML concepts, and the use of policy sets to configure and administer web services settings. Complete the following actions before you begin this task:

- Read about propagating self-issued SAML holder-of-key tokens with asymmetric key by using WSS APIs.
- Become familiar with using embedded key materials in SAML tokens for message protection by using WSS APIs. Your usage scenario requires requesting SAML tokens from an external STS instead of using self-issued SAML tokens.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with message level protection.
- Read about requesting SAML sender-vouches tokens from an external STS to propagate by using WSS APIs with transport level protection.
- Read about requesting SAML bearer tokens from an external STS, which you propagate by using WSS APIs with transport level protection.
- Become familiar with accessing an external STS by using WSS APIs.

About this task

This task shows example code to request SAML tokens with the holder-of-key subject confirmation method and the embedded public key in an X.509 certificate by using WSS APIs, from an external STS. This task focuses on sending an X.509 certificate to an external STS when requesting SAML holder-of-key tokens.

Procedure

1. Specify an STS from which to request a SAML security token that contains holder-of-key subject confirmation method; for example:

```
com.ibm.websphere.wssecurity.wssapi.WSSFactory factory =
    com.ibm.websphere.wssecurity.wssapi.WSSFactory.getInstance();
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(SamlConstants.STS_ADDRESS, "https://www.example.com/sts");
cbackMap1.put(SamlConstants.SAML_APPLIES_TO, "http://myhost:9080/myService");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_SOAP_VERSION, "1.1");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
              "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST,
              "true");   //RST or RSTC
cbackMap1.put(SamlConstants.TOKEN_TYPE,
              "http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0");
cbackMap1.put(SamlConstants.CONFIRMATION_METHOD, "holder-of-key");
```

For the holder-of-key subject confirmation method, you must specify whether a public key or a symmetric key is embedded in SAML tokens. This example specifies a public key type. It then specifies the location of a certificate that contains the public key, and the location of the corresponding private key for the sender to digitally sign elements of SOAP messages to satisfy the holder-of-key subject confirmation requirements.

2. Specify the location of an X.509 certificate to embed in SAML tokens and a corresponding private key for using to digitally sign message elements; for example:

```
cbackMap1.put(SamlConstants.KEY_TYPE,
              "http://docs.oasis-open.org/ws-sx/ws-trust/200512/PublicKey");
cbackMap1.put(SamlConstants.KEY_ALIAS, "soapinitiator" );
cbackMap1.put(SamlConstants.KEY_NAME, "CN=SOAPInitator, O=Example");
cbackMap1.put(SamlConstants.KEY_PASSWORD, "keypass");
cbackMap1.put(SamlConstants.KEY_STORE_PATH, "keystores/initiator.jceks");
cbackMap1.put(SamlConstants.KEY_STORE_PASSWORD, "storepass");
cbackMap1.put(SamlConstants.KEY_STORE_TYPE, "jceks");

SAMLGenerateCallbackHandler cbHandler1 = new SAMLGenerateCallbackHandler(cbackMap1);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

SecurityToken samlToken = factory.newSecurityToken(SAMLToken.class,
                                                   cbHandler1, "system.wss.generate.saml");
```

The specified X.509 certificate is sent in WS-Trust requests to the external STS in the `trust:UseKey` element. For more information read about SAML assertions defined in the SAML Token Profile standard. SSL is used to protect integrity and confidentiality of WS-Trust request and response messages in this example.

Results

You have learned key building blocks to request SAML tokens with the holder-of-key subject confirmation method and asymmetric key from an external STS using WSS APIs. To use the SAML token to sign request messages, become familiar with the example code in the "Propagating self-issued SAML holder-of-key tokens with asymmetric key by using WSS APIs" topic.

Sending a security token using WSSAPIs with a generic security token login module:

You can request an authentication token from an external *Security Token Service (STS)*, and then send the token with web service request messages using the Java API for XML-Based Web Services (JAX-WS) programming model and Web Services Security APIs (WSS API), with message or transport level protection.

Before you begin

This task assumes that you are familiar with the JAX-WS programming model, the WSS API interfaces, WebSphere web service security generic security token login modules, SSL transport protection, message level protection, and the use of policy sets to configure and administer web services settings.

About this task

The web service client application used in this task is a modified version of the client code that is contained in the JaxWSServicesSamples sample application that is available for download. Code examples from the sample are described in the procedure, and a complete, ready-to-use web service client sample is provided.

Complete the following steps to request a SAML Bearer authentication token from an external STS and send the token:

Procedure

1. Identify and obtain the web service client that you want to use to invoke a web service provider. Use this client to request and to insert authentication tokens in the SOAP request messages programmatically using WSS APIs. The web service client used in this procedure is a modified version of the client code that is contained in the JaxWSServicesSamples web service sample application.

   Complete the following steps to obtain and modify the sample web service client to add the Web Services Security API to pass a security token in the SOAP request message programmatically using WSS APIs:

   a. Download the JaxWSServicesSamples sample application. The JaxWSServicesSamples sample is not installed by default.

   b. Obtain the JaxWSServicesSamples client code. For the purpose of this example, the procedure uses a modified version of the Echo thin client sample that is included in the JaxWSServicesSamples sample. The web service Echo thin client sample file, `SampleClient.java`, is located in the `src\SampleClientSei\src\com\ibm\was\wssample\sei\cli` directory. The sample class file is included in the `WSSampleClientSei.jar` file.

      The `JaxWSServicesSamples.ear` enterprise application and supporting Java archives (JAR) files are located in the `installableApps` directory within the JaxWSServicesSamples sample application.

   c. Deploy the `JaxWSServicesSamples.ear` file onto the application server. After you deploy the `JaxWSServicesSamples.ear` file, you are ready to test the sample web service client code against the sample application.

      Instead of using the PolicySet for the protection of the web service client sample, you can choose to add the code snippets to pass authentication tokens in the SOAP request message programmatically using WSS APIs in your own web service client application. The example in this procedure uses a JAX-WS web service thin client; however, you can also use a managed client.

   d. Attach the SAML11 Bearer WSHTTPS default policy set to the web services provider. This policy set is used to protect messages using HTTPS transport. Read about configuring client and provider bindings for the SAML Bearer token for details on how to attach the SAML11 Bearer WSHTTPS default policy set to the web services provider.

   e. Assign the SAML Bearer Provider sample default general bindings to the sample web services provider. Read about configuring client and provider bindings for the SAML bearer token for details on assigning the SAML Bearer Provider sample default general bindings to your web services application.

   f. Verify that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate. Complete the following steps by using the administrative console:

      1) Click **Services** > **Policy sets** > **General provider policy set bindings** > **Saml Bearer Provider sample** > **WS-Security** > **Authentication and protection**.

      2) Click **con_saml11token** in the Authentication tokens table.

      3) Click **Callback handler**.

      4) In the Custom Properties section, ensure that the trustStoreType, trustStorePassword and trustStorePath custom properties correspond to the trust store containing the STS signer certificate.

2. If you are using SSL Transport-level protection to protect the web service request or the WS-Trust request, use the following Java virtual machine (JVM) property to set up the SSL configuration.

```
-Dcom.ibm.SSL.ConfigURL=file:<profile_root>\properties\ssl.client.props
```

   Alternatively, you can define the SSL configuration file using a Java system property in the sample client code:

```
System.setProperty("com.ibm.SSL.ConfigURL", "file:profile_root/properties/ssl.client.props");
```

3. Add the JAR file for the JAX-WS thin client to the class path: `app_server_root/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar`. See the testing web service-enabled clients information for more information about adding this JAR file to the class path.
4. Request the authentication token from an external STS. The following code snippet illustrates how to request the authentication token to be used with WebSphere generic SecurityToken login module, and assumes that an external STS is configured to accept a Username token as authentication token, and to issue a SAML 1.1 token.

```
//Request SecurityToken from external STS:
WSSFactory factory = WSSFactory.getInstance();
//STS URL that issues the requested token
String STS_URI  = "https://externalstsserverurl:port/TrustServerWST13/services/RequestSecurityToken";
//Web services endpoint that receives the issued token
String ENDPOINT_URL = "http://localhost:9080/WSSampleSei/EchoService";

//Begin sample code 1 (Using WS-Trust Issue to request the token from
//the STS in which authentication token is send over WS-Security head):
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(IssuedTokenConfigConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(IssuedTokenConfigConstants.APPLIES_TO, ENDPOINT_URL);
//The following property specifies that the ws-trust request should be
//compliance with WS-Trust 1.3 spec
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
"http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
//This request is made with WS-TRust Issue only (without the use of
//WS-Trust Validate)
cbackMap1.put(IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT, "false");

GenericIssuedTokenGenerateCallbackHandler cbHandler1 =
new GenericIssuedTokenGenerateCallbackHandler (cbackMap1);

//Create the context object for WS-Trust request:
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
// Use UNT for trust request authentication
UNTGenerateCallbackHandler utCallbackHandler = new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
gencont1.add(ut);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);
//End of sample code 1.

//Begin sample code 2 (using WS-Trust Validate to request a token by
//exchanging a token in RunAs Subject).
//If web service client has RunAs Subject , for example an
//authenticated intermediate server acts as a client to invoke the
//downstream service, you can program the client to use the token from
//the RunAs subject to exchange with the STS by using WS-Trust validate.
//To do so, you replace sample code 1 with the following:
HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
cbackMap1.put(IssuedTokenConfigConstants.STS_ADDRESS, STS_URI);
cbackMap1.put(IssuedTokenConfigConstants.APPLIES_TO, ENDPOINT_URL);
//This request is made with WS-Trust 1.3
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
"http://docs.oasis-open.org/ws-sx/ws-trust/200512");
cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");

//add the next line if you do not want to fallback to WS-Trust Issue if
//token exchange fails.
cbackMap1.put(IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT_ONLY, "true");

//add the next line to specify the token type in the RunAs subject that
//will be used to exchange the requested token. For example, you use
//the LTPA token to exchange for a SAML token. If the exchanged token
//in the RunAs subject has the same value type as the requested token,
//setting IssuedTokenConfigConstants.USE_TOKEN is not required.
cbackMap1.put(IssuedTokenConfigConstants.USE_TOKEN, LTPAToken.ValueType);

GenericIssuedTokenGenerateCallbackHandler cbHandler1 =
    new GenericIssuedTokenGenerateCallbackHandler (cbackMap1);
//The following codes are added if Authentication token in ws-security
//head or Message level security protection is required. If there is no
//Message level protection or additional authentication token for
//WS-Trust Validate, do not create the context object shown below.
//Context object for WS-Trust request:
```

```
WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
WSSConsumingContext concont1 = factory.newWSSConsumingContext();
// Use UNT for trust request authentication
UNTGenerateCallbackHandler utCallbackHandler =
    new UNTGenerateCallbackHandler("testuser", "testuserpwd");
SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
gencont1.add(ut);
cbHandler1.setWSSConsumingContextForTrustClient(concont1);
cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

//End of sample code 2.

GenericSecurityToken token = (GenericSecurityToken) factory.newSecurityToken
(GenericSecurityToken.class, cbHandler1, "system.wss.generate.issuedToken");

//The following step to set ValueType is required..
//The parameter is always the QName of the requested token's valueType.
//QName for SAML1.1:
QName Saml11ValueType = new QName(WSSConstants.SAML.SAML11_VALUE_TYPE);
            token.setValueType(Saml11ValueType);

//This article includes QName definitions for  SAML11, SAML20, TAM
//token, and Pass ticket token.
//QName for SAML 2.0:
QName Saml20ValueType = new QName(WSSConstants.SAML.SAML20_VALUE_TYPE);
token.setValueType(Saml11ValueType);
//QName for TAM token:
QName TamValueType = new QName("http://ibm.com/2004/01/itfim/ivcred");
//QName for PassTicket token:
QName PassTicketValueType = new QName("http://docs.oasis-open.org/wss/
2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken");

//You can use the Token interface to get the token ValueType QName for
//all other tokens. For example, a Username Token's QName is //UsernameToken.ValueType.
```

The `GenericIssuedTokenGenerateCallbackHandler` object contains parameters that define the characteristics of the security token that you are requesting, as well as other parameters required to reach the STS and to obtain the security token. The `GenericIssuedTokenGenerateCallbackHandler` object specifies the configuration parameters described in the following table:

Table 207. `GenericIssuedTokenGenerateCallbackHandler` properties. This table describes the configuration parameters for the `GenericIssuedTokenGenerateCallbackHandler` object, and specifies whether or not the property is required.

Property	Description	Required
`IssuedTokenConfigConstants.STS_ADDRESS`	Specifies the http address of the STS. When communication to the STS is protected with SSL, you must set the `-Dcom.ibm.SSL.ConfigURL` property. SSL connection to the STS is indicated with an `https://` address prefix.	Yes
`IssuedTokenConfigConstants.APPLIES_TO`	Specifies the target service address for where you want to use the token.	No
`IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST`	Specifies whether to request a single token from the STS that is enclosed in a `RequestSecurityToken (RST)` element or multiple tokens in a collection of RST elements that are enclosed in a single `RequestSecurityTokenCollection (RSTC)` element. The default behavior is to request a single token that is enclosed in a `RequestSecurityToken (RST)` element from the STS. Specifying a `true` value for this property indicates a request for multiple tokens in a collection of RST elements that are enclosed in a single `RequestSecurityTokenCollection (RSTC)` element from the STS. The default value is `false`.	No
`IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE`	Specifies the WS-Trust namespace that is included in the WS-Trust request. The default value is `WSTrust 1.3`.	No

Table 207. `GenericIssuedTokenGenerateCallbackHandler` properties (continued). This table describes the configuration parameters for the `GenericIssuedTokenGenerateCallbackHandler` object, and specifies whether or not the property is required.

Property	Description	Required
`IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT`	Specify if you want WS-Security to use the token from the RunAs subject to exchange the requested token first by using WS-Trust Validate. If set to `false`, WS-Security will use WS-Trust Issue to request the token. The default value is `true`.	No
`IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT_ONLY`	Specify if you do not want WS-Security to use WS-Trust Issue to the requested token if token exchange fails. The default value is `false`.	No
`IssuedTokenConfigConstants.USE_TOKEN`	Use this value to choose a token from the RunAs subject to exchange the requested token. The default value is the requested token's `ValueType`.	No

A `WSSGenerationContext` instance and a `WSSConsumingContext` instance are also set in the `GenericIssuedTokenGenerateCallbackHandler` object. In this example, the `WSSGenerationContext` instance contains a `UNTGenerateCallbackHandler` object with the information to create the `UsernameToken` that you want to send to the STS.

The `system.wss.generate.issuedToken` parameter specifies the Java Authentication and Authorization Service (JAAS) login module that is used to create the generic security token. You must specify a JVM property to define a JAAS configuration file that contains the required JAAS login configuration; for example:

```
-Djava.security.auth.login.config=profile_root/properties/wsjaas_client.conf
```

Alternatively, you can specify a JAAS login configuration file by setting a Java system property in the sample client code; for example:

```
System.setProperty("java.security.auth.login.config", "profile_root/properties/wsjaas_client.conf");
```

5. Add the requested authentication token from the STS to the SOAP security header of web services request messages.
   a. Initialize the web service client and configure the `SOAPAction` properties. The following code illustrates these actions:

```
// Initialize web service client
EchoService12PortProxy echo = new EchoService12PortProxy();
echo._getDescriptor().setEndpoint(endpointURL);

// Configure SOAPAction properties
BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
Map<String, Object> requestContext = bp.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(token);
```

   b. Initialize the `WSSGenerationContext` object. The following code illustrates using the `WSSFactory.newWSSGenerationContext` method to obtain a `WSSGenerationContext` object. The `WSSGenerationContext` object is then used to insert the token into a web service request message:

```
// Initialize WSSGenerationContext
WSSGenerationContext gencont = factory.newWSSGenerationContext();
gencont.add(token);
```

The `WSSGenerationContext.add` method requires the client code to have the following Java 2 Security permission:

```
permission javax.security.auth.AuthPermission "modifyPrivateCredentials"
```

6. Add an X.509 token for message protection (skip this step if the web service is protected with SSL Transport level protection only). The following sample code uses the `dsig-sender.ks` key file and the

SOAPRequester sample key. You must not use the sample key in a production environment. The following code illustrates adding an X.509 token for message protection:

```
//Add an X.509 Token for message protection
X509GenerateCallbackHandler x509callbackHandler = new X509GenerateCallbackHandler(
    null,
    "profile_root/etc/ws-security/samples/dsig-sender.ks",
    "JKS",
    "client".toCharArray(),
    "soaprequester",
    "client".toCharArray(),
    "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);

SecurityToken x509 = factory.newSecurityToken(X509Token.class,
x509callbackHandler, "system.wss.generate.x509");

WSSSignature sig = factory.newWSSSignature(x509);
sig.setSignatureMethod(WSSSignature.RSA_SHA1);

WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(token);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
sig.addSignPart(sigPart);
sig.addSignPart(WSSSignature.BODY);
```

a. Create a WSSSignature object with the X.509 token. The following line of code creates a WSSSignature object with the X.509 token:

```
WSSSignature sig = factory.newWSSSignature(x509);
```

b. Add the signed part to use for message protection. The following line of code specifies to add WSSSignature.BODY as the signed part:

```
sig.addSignPart(WSSSignature.BODY);
```

c. Add the Timestamp element in the SOAP Security header. The SAML20 SenderVouches WSHTTPS and SAML11 SenderVouches WSHTTPS policy sets require web service requests and responses to contain a Timestamp element in the SOAP Security header. In the following code, the WSSFactory.newWSSTimestamp() method generates a Timestamp element, and the WSSGenerationContext.add(timestamp) method adds the Timestamp element to the request message:

```
// Add Timestamp element
WSSTimestamp timestamp = factory.newWSSTimestamp();
    gencont.add(timestamp);
sig.addSignPart(WSSSignature.TIMESTAMP);

gencont.add(sig);

WSSConsumingContext concont = factory.newWSSConsumingContext();
```

d. Skip this step if token signature is not required. If the requested security token needs to be signed with the *STR Dereference Transform* reference option, follow step 1. Otherwise, follow step 2. The *STR Dereference Transform* reference option is commonly known as *STR-Transform*.

Step 1: Some tokens, including SAML Tokens, cannot be digitally signed directly. You must sign the token using *STR-Transform*. In order for a token to be signed with *STR-Transform*, it must be referenced by a <wsse:SecurityTokenReference> element in the <wsse:Security> header block. To sign a security token with *STR-Transform*, a separate WSSSignPart is created to specify the SecurityTokenReference with a transformation algorithm that is represented by the WSSSignPart.TRANSFORM_STRT10 attribute. This attribute enables the WS-Security runtime environment to generate a SecurityTokenReference element to reference the token, and to digitally sign the token using the *STR Dereference* reference option. The following code illustrates the use of the WSSSignPart.TRANSFORM_STRT10 attribute:

```
WSSSignPart sigPart = factory.newWSSSignPart();
sigPart.setSignPart(token);
sigPart.addTransform(WSSSignPart.TRANSFORM_STRT10);
```

Step 2: If the requested signed token is not a SAML token, or *STR-Transform* is not used, use the following code instead:

```
sig.addSignPart(token);
```

e. Attach the WSSGenerationContext object to the web service RequestContext object. The WSSGenerationContext object now contains all of the security information required to format a request message. The WSSGenerationContext.process(requestContext) method attaches the

WSSGenerationContext object to the web service RequestContext object to enable the WS-Security runtime environment to format the required SOAP Security header; for example:

```
// Attaches the WSSGenerationContext object to the web service RequestContext object.
gencont.process(requestContext);
```

7. Use the X.509 token to validate the digital signature and the integrity of the response message if the provider policy requires the response message to be digitally signed. Skip this step if using SSL Transport level protection.
   a. An X509ConsumeCallbackHandler object is initialized with a trust store and a List of certificate path objects to validate the digital signature in a response message. The following code initializes the X509ConsumeCallbackHandler object with dsig-receiver.ks trust store and a certificate path object called certList:

```
ArrayList certList = new ArrayList();
java.security.cert.CertStore certStore = java.security.cert.CertStore.getDefaultType();
certList.add(certStore);

X509ConsumeCallbackHandler callbackHandlerVer = new
    X509ConsumeCallbackHandler("profile_root/etc/ws-security/samples/dsig-receiver.ks",
    "JKS",
    "server".toCharArray(),
    certList,
    java.security.Security.getProvider("IBMCertPath"));
```

   b. A WSSVerification object is created and the message body is added to the verification object so that the WS-Security runtime environment validates the digital signature. The following code is used to initialize the WSSVerification object:

```
WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandlerVer);
```

The WSSConsumingContext object now contains all the security information that is required to format a request message. The WSSConsumingContext.process(requestContext) method attaches the WSSConsumingContext object to the response method; for example:

```
// Attaches the WSSConsumingContext object to the web service RequestContext object.
concont.process(requestContext);
```

Results

You have requested a security token from an external STS. After obtaining the token, you sent the token with web services request messages using message level protection using the JAX-WS programming model and WSS APIs.

Example

The following code example is a web service client application that demonstrates how to request a SAML Bearer token from an external STS and send that SAML token in a web service request message. If your usage scenario requires SAML tokens, but does not require your application to pass the SAML tokens using web service messages, you only need to use the first part of the following sample code, up through the // Initialize web service client section.

```
package com.ibm.was.wssample.sei.cli;
import com.ibm.was.wssample.sei.echo.EchoService12PortProxy;
import com.ibm.was.wssample.sei.echo.EchoStringInput;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
import com.ibm.websphere.wssecurity.wssapi.WSSTimestamp;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.wssapi.token.UsernameToken;
import com.ibm.websphere.wssecurity.callbackhandler.UNTGenerateCallbackHandler;
import com.ibm.wsspi.wssecurity.core.token.config.WSSConstants;
import com.ibm.wsspi.wssecurity.core.config.IssuedTokenConfigConstants;
import com.ibm.websphere.wssecurity.callbackhandler.GenericIssuedTokenGenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.token.GenericSecurityToken;
import javax.xml.namespace.QName;
import java.util.HashMap;
import java.util.Map;

import javax.xml.ws.BindingProvider;

public class SampleSamlSVClient {
    private String urlHost = "yourhost";
    private String urlPort = "9444";
```

```
private static final String CONTEXT_BASE = "/WSSampleSei/";
private static final String ECHO_CONTEXT12 = CONTEXT_BASE+"EchoService12";
private String message = "HELLO";
private String uriString = "https://" + urlHost + ":" + urlPort;
private String endpointURL = uriString + ECHO_CONTEXT12;
private String input = message;

/**
 * main()
 *
 * see printusage() for command-line arguments
 *
 * @param args
 */
public static void main(String[] args) {
    SampleSamlSVClient sample = new SampleSamlSVClient();
    sample.CallService();
}

/**
 * CallService Parms were already read. Now call the service proxy classes
 *
 */
void CallService() {
    String response = "ERROR!:";
    try {
        System.setProperty("java.security.auth.login.config",
            "file:/opt/IBM/WebSphere/AppServer/profiles/AppSrv01/properties/wsjaas_client.conf ");
        System.setProperty("com.ibm.SSL.ConfigURL",
            "file:/opt/IBM/WebSphere/AppServer/profiles/AppSrv01/properties/ssl.client.props");

        //Request the SAML Token from external STS
        WSSFactory factory = WSSFactory.getInstance();
        String STS_URI = "https://yourhost:9443/TrustServerWST13/services/RequestSecurityToken";
        String ENDPOINT_URL = "http://localhost:9081/WSSampleSei/EchoService12";

        HashMap<Object, Object> cbackMap1 = new HashMap<Object, Object>();
        cbackMap1.put(IssuedTokenConfigConstants.STS_ADDRESS, STS_URI);
        cbackMap1.put(IssuedTokenConfigConstants.APPLIES_TO, ENDPOINT_URL);
        cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_WSTRUST_NAMESPACE,
            "http://docs.oasis-open.org/ws-sx/ws-trust/200512");
        cbackMap1.put(IssuedTokenConfigConstants.TRUST_CLIENT_COLLECTION_REQUEST, "false");
        cbackMap1.put(IssuedTokenConfigConstants.USE_RUN_AS_SUBJECT, "false");

        GenericIssuedTokenGenerateCallbackHandler cbHandler1 =
            new GenericIssuedTokenGenerateCallbackHandler (cbackMap1);

        //Context object for WS-Trust request:
        WSSGenerationContext gencont1 = factory.newWSSGenerationContext();
        WSSConsumingContext concont1 = factory.newWSSConsumingContext();

        // Use UNT for trust request authentication
        UNTGenerateCallbackHandler utCallbackHandler = new
            UNTGenerateCallbackHandler("testuser", "testuserpwd");
        SecurityToken ut = factory.newSecurityToken(UsernameToken.class, utCallbackHandler);
        gencont1.add(ut);
        cbHandler1.setWSSConsumingContextForTrustClient(concont1);
        cbHandler1.setWSSGenerationContextForTrustClient(gencont1);

        //get generic security token
        GenericSecurityToken token = (GenericSecurityToken) factory.newSecurityToken
            (GenericSecurityToken.class, cbHandler1, "system.wss.generate.issuedToken");
        QName Saml11ValueType = new QName(WSSConstants.SAML.SAML11_VALUE_TYPE);
        token.setValueType(Saml11ValueType);
        System.out.println("SAMLToken id = " + token.getId());

        // Initialize web services client
        EchoService12PortProxy echo = new EchoService12PortProxy();
        echo._getDescriptor().setEndpoint(endpointURL);

        // Configure SOAPAction properties
        BindingProvider bp = (BindingProvider) (echo._getDescriptor().getProxy());
        Map<String, Object> requestContext = bp.getRequestContext();
        requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpointURL);
        requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
        requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, "echoOperation");

        // Initialize WSSGenerationContext
        WSSGenerationContext gencont = factory.newWSSGenerationContext();
        gencont.add(token);

        // Add timestamp
        WSSTimestamp timestamp = factory.newWSSTimestamp();
        gencont.add(timestamp);
        gencont.process(requestContext);

        // Build the input object
        EchoStringInput echoParm =
            new com.ibm.was.wssample.sei.echo.ObjectFactory().createEchoStringInput();
        echoParm.setEchoInput(input);
```

```
            System.out.println(">> CLIENT: SEI Echo to " + endpointURL);

            // Prepare to consume timestamp in response message.
            WSSConsumingContext concont = factory.newWSSConsumingContext();
            concont.add(WSSConsumingContext.TIMESTAMP);
            concont.process(requestContext);

            // Call the service
            response = echo.echoOperation(echoParm).getEchoResponse();

            System.out.println(">> CLIENT: SEI Echo invocation complete.");
            System.out.println(">> CLIENT: SEI Echo response is: " + response);
        } catch (Exception e) {
            System.out.println(">> CLIENT: ERROR: SEI Echo EXCEPTION.");
            e.printStackTrace();
        }
    }
}
```

Securing messages at the response consumer using WSS APIs:

You can secure SOAP messages with signature verification, decryption, and consumer tokens to protect message integrity, confidentiality, and authenticity, respectively. The response consumer (client-side) configuration defines the Web Services Security requirements for the incoming SOAP response.

About this task

To secure web services with WebSphere Application Server, you must configure the generator and the consumer security constraints. You must specify several different configurations. Although there is no specific sequence to specify these different configurations, some configurations reference other configurations. For example, decryption configurations reference encryption configurations.

The response consumer (client-side) configuration requirements involve verifying that the integrity parts are signed and that the signature is verified, verifying that the required confidential parts are encrypted and that the parts are decrypted; and validating the security tokens.

You can use the following methods to configure Web Services Security and to define policy types to secure the SOAP messages:

- Use the administrative console to configure policy sets.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

The following high-level steps use the WSS APIs:

Procedure

- Verify consumer signing information to protect message integrity.
- Configure decryption to protect message confidentiality.
- Validate consumer tokens to protect message authenticity.

Results

After completing these procedures, you have secured messages at the response consumer level.

What to do next

Next, if not already configured, secure messages with signing information, encryption, and generator tokens at the response (client-side) generator level.

Configuring decryption methods to protect message confidentiality using the WSS APIs:

You can configure decryption method information for the response consumer (client side) section of the binding file. Decryption information is used to specify how the consumers (receivers) decrypt incoming

SOAP messages. To configure decryption, specify which message parts to decrypt and specify which algorithm methods and security tokens are to be used for decryption.

Before you begin

Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. Prior to configuring decryption, familiarize yourself with XML encryption.

About this task

For decryption, you must specify the following:

- Which parts of the message are to be decrypted.
- Which decryption algorithms to specify.

To configure decryption and decrypted parts on the client side, use the WSSDecryption and WSSDecryptPart APIs, or configure policy sets using the administrative console.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

WebSphere Application Server uses decryption information for the default consumer to decrypt parts of the SOAP message. The WSSDecryption API configures the following required parts as decrypted parts.

Table 208. Required decrypted parts. Use the decryption information to specify how incoming messages are decrypted.

Decryption parts	Description
Keywords	Keywords are used to add the decrypted parts to the SOAP message.
XPath expression	XPath expressions are used to add the decrypted parts to the SOAP message.
WSSDencryptPart object	This object adds the decrypted parts to the SOAP message.
WSSVerification object	This object adds the signature verification component as a decrypted part.
Header	This part adds the header in the SOAP header, specified by QName, as a decrypted part.
Security token object	This object adds the security token as a decrypted part.

Web Services Security API (WSS API) supports symmetric encryption, by using a shared key, only when Web Services Secure Conversation (WS-SecureConversation) is used.

The WSS APIs allow the use of either keywords or an XPath expression to specify the parts of the SOAP message that are to be decrypted. WebSphere Application Server supports the use of the following keywords:

Table 209. Supported decryption keywords. Use the keywords to decrypt incoming messages.

Keyword	References
BODY_CONTENT	The keyword for the body contents of the SOAP message body as a decryption target.
SIGNATURE	The keyword for the signature element as a decryption target.
USERNAME_TOKEN,	The keyword for the Username token element as a decryption target.

If configuring using the WSS APIs, the WSSDecryption and WSSDecryptPart APIs complete these high-level steps:

Procedure

1. Use the WSSDecryption API to configure encryption. The WSSDecryption API performs these tasks by default:

a. Generates the callback handler.
b. Generates the consumer security token object.
c. Adds the security token reference type.
d. Adds the WSSEncryptPart object.
e. Adds the parts to be encrypted. Adds the default parts for decryption by using keywords and XPath expressions.
f. Adds the verification component.
g. Adds the header in the SOAP message, specified by QName.
h. Sets the default data encryption method.
i. Specifies whether the key is to be decrypted using a Boolean value. Calls this method when the shared key is encrypted.
j. Sets the default key encryption method.

2. Use the WSSEncryptPart API to configure encrypted parts or add a transform method. The WSSEncryptPart API performs these tasks by default:
   a. Sets the encrypted parts specified by using keywords or an XPath expression.
   b. Sets the encrypted parts specified by an XPath expression.
   c. Sets the signature component object, WSSSignature.
   d. Sets the header in the SOAP message, specified by QName.
   e. Sets the generator security token.
   f. Adds the transform method, if needed.
3. Change from the default values for algorithm or message parts, as needed. For example: you could change one or more of the following items:
   - Add USERNAME_TOKEN as a target of decryption.
   - Change the data encryption algorithm from the default value of AES 128.
   - Change the key encryption algorithm from the default value of KW_RSA_OAEP.
   - Specify to not encrypt the encryption key (false).
   - Change the security token type from the default value of X.509 token.
   - Only use BODY_CONTENT as an encryption part and not use SIGNATURE also.

Results

The decryption information is configured for the consumer binding.

Example

The following is an example of the WSSDecryption API:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();
    X509ConsumeCallbackHandler callbackhandler = generateCallbackHandler();
// see X509ConsumeCallbackHandler
    WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
        callbackhandler);

 concont.add(dec);
```

What to do next

You must configure similar encryption information for the client-side request generator (sender) bindings, if you have not already configured the information.

Next, review the WSSDecryption API process.

Decrypting SOAP messages using the WSSDecryption API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure the client for decryption on the response (client) consumer side, use the WSSDecryption API to decrypt the SOAP messages. The WSSDecryption API specifies which request SOAP message parts to decrypt when configuring the client.

Before you begin

You can use the WSS API or use policy sets on the administrative console to enable decryption and add consumer security tokens in the SOAP message. To secure SOAP messages, you must have completed the following decryption tasks:

- Encrypted the SOAP message.
- Chosen the decryption method.

About this task

The decryption information on the consumer side is used for decrypting an incoming SOAP message for the response consumer (client side) bindings. The client consumer configuration must match the configuration for the provider generator.

Confidentiality settings require that confidentiality constraints be applied to generated messages.

The following decryption parts can be configured:

Table 210. Decryption parts. Use the decryption parts to secure messages.

Decryption parts	Description
part	Adds the WSSDecryptPart object as a target of the decryption part.
keyword	Adds the decryption part using keywords. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE • USERNAME_TOKEN
xpath	Adds the decryption part using an XPath expression.
verification	Adds the WSSVerification instance as a target of the decryption part.
header	Adds the SOAP header, specified by QName, as a target of the decryption part.

For decryption, certain default behaviors occur. The simplest way to use the WSS API for decryption is to use the default behavior (see the example code). WSSDecryption provides defaults for the key encryption algorithm, the data encryption algorithm, and the decryption parts such as the SOAP body content and the signature. The decryption default behaviors include:

Table 211. Decryption decisions. Several decryption part characteristics are configured by default.

Decryption decisions	Default behavior
Which parts to decrypt	The default decryption parts are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports using these keywords: • WSSDecryption.BODY_CONTENT • WSSDecryption.SIGNATURE • WSSDecryption.USERNAME_TOKEN After you specify which message parts to decrypt, you must specify which method to use when decrypting the consumer request message. For example, if both signature and body content are applied for encryption, then the SOAP message parts that are decrypted include the same parts.
Whether to encrypt the key (isEncrypt)	The default value is to encrypt the key (`true`).

Table 211. Decryption decisions (continued). Several decryption part characteristics are configured by default.

Decryption decisions	Default behavior
Which data decryption algorithm to choose (method)	The default data decryption algorithm method is AES128. WebSphere Application Server supports these data encryption methods: • WSSDecryption.AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc • WSSDecryption.AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc • WSSDecryption.AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc • WSSDecryption.TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc
Which key decryption method to choose (algorithm)	The default key decryption algorithm method is key wrap RSA OAEP. WebSphere Application Server supports these key encryption methods: • WSSDecryption.KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128 • WSSDecryption.KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192 • WSSDecryption.KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256 • WSSDecryption.KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p • WSSDecryption.KW_RSA15: http://www.w3.org/2001/04/xmlenc#rsa-1_5 • WSSDecryption.KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes
Which security token to specify	The default security token type is the X509 token. WebSphere Application Server provides the following pre-configured consumer token types: • Derived key token • X509 tokens

Procedure

1. To decrypt the SOAP message using the WSSDecryption API, first ensure that the application server is installed.
2. The WSS API process for decryption performs these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. The WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates the callback handler for the consumer side.
   d. Creates WSSDecryption with the class for the security token and the callback handler from the WSSFactory instance. The default behavior of WSSDecryption is to assume that the body content and the signature are encrypted.
   e. Adds the parts to be decrypted, if the default is not appropriate.
   f. Adds the candidates of the data encryption methods to use for decryption.
   g. Adds the candidates of the key encryption methods to use for decryption.
   h. Adds the candidates of the security token to use for decryption.
   i. Calls WSSDecryption.encryptKey(false) if the application does not want the key to be encrypted in the incoming message.
   j. Adds WSSDecryption to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAPMessageContext

Results

If there is an error condition during decryption, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code for decrypting the SOAP message body content:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();
```

```
// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
       X509ConsumeCallbackHandler(
                                  "",
                                  "enc-sender.jceks",
                                  "jceks",
                                  "storepass".toCharArray(),
                                  "alice",
                                  "keypass".toCharArray(),
 "CN=Alice, O=IBM, C=US");

// Generate the WSSDecryption instance (step: d)
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                               callbackHandler);

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part to be encrypted (step: e)
// DEFAULT: WSSEncryption.BODY_CONTENT and WSSEncryption.SIGNATURE

// Set the part specified by the keyword (step: e)
      dec.addRequiredDecryptPart(WSSDecryption.BODY_CONTENT);

// Set the part in the SOAP Header specified by QName (step: e)
    dec.addRequiredDecryptHeader(new
       QName("http://www.w3.org/2005/08/addressing",
       "MessageID"));

// Set the part specified by WSSVerification  (step: e)
   X509ConsumeCallbackHandler verifyCallbackHandler =
      getCallbackHandler();
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                    verifyCallbackHandler);
      dec.addRequiredDecryptPart(ver);

// Set the part specified by XPath expression (step: e)
      StringBuffer sb = new StringBuffer();
         sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Envelope']");
         sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Body']");
         sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Ping']");
         sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Text']");
      dec.addRequiredDecryptPartByXPath(sb.toString());

// Set the part in the SOAP header to be decrypted specified by QName (step: e)
      dec.addRequiredDecryptHeader(new
          QName("http://www.w3.org/2005/08/addressing",
          "MessageID"));

// Set the candidates for the data encryption method (step: f)
// DEFAULT : WSSDecryption.AES128
      dec.addAllowedEncryptionMethod(WSSDecryption.AES128);
      dec.addAllowedEncryptionMethod(WSSDecryption.AES192);

// Set the candidates for the key encryption method (step: g)
// DEFAULT : WSSDecryption.KW_RSA_OAEP
      dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);

// Set the candidate security token to used for the decryption (step: h)
   X509ConsumeCallbackHandler callbackHandler2 = getCallbackHandler2();
      dec.addToken(X509Token.class, callbackHandler2);

// Set whether or not the key should be encrypted in the incoming SOAP message (step: i)
// DEFAULT: true
      dec.encryptKey(true);

// Add the WSSDecryption to the WSSConsumingContext (step: j)
   concont.add(dec);

// Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

What to do next

Next, use the WSSDecryptPart API or configure the policy sets using the administrative console to add decrypted parts for the consumer message.

Choosing decryption methods for the consumer binding:

To configure the client for response decryption for the consumer binding, specify which data and transform algorithm methods to use when the client decrypts the SOAP messages.

Before you begin

Prior to completing these steps, read the XML encryption information to become familiar with encrypting and decrypting SOAP messages.

To complete decryption configuration to secure SOAP messages, you must complete the following tasks:

- Configure decryption of the SOAP message parts
- Specify the decryption methods.

You can configure the decryption methods using the WSSDecryption and WSSDecryptPart APIs. Or you can also configure policy sets using the administrative console to configure the decryption methods.

About this task

Some of the encryption-related definitions are based on the XML-Encryption specification. The following information defines some data encryption-related terms:

Data encryption method algorithm

Data encryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. This algorithm encrypts and decrypts data in fixed size, multiple octet blocks.

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key encryption method algorithm

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. The algorithm represents public key encryption algorithms that are specified for encrypting and decrypting keys.

By default, the RSA_OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method:

- http://www.w3.org/2001/04/xmlenc#sha256
- http://www.w3.org/2001/04/xmlenc#sha512

By default, the RSA_OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the KW_AES256 and the KW_AES192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

To complete the decryption configuration, you must specify the algorithm uniform resource identifier (URI) and its usage type. If the URI is used for multiple usage types, then you must define the URI to each usage type. WebSphere Application Server supports the following decryption usage types:

Table 212. Decryption usage types. These decryption types are supported by WebSphere Application Server.

Usage types	Description
Data encryption	Specifies the algorithm URI that is used for both encrypting and decrypting data. Encrypts and decrypts data in fixed size, multiple octet blocks.
Key encryption	Specifies the algorithm URI that is used for encrypting and decrypting the encryption key.

To configure the decryption and decrypted part algorithms, use the WSSDecryption and WSSDecryptPart APIs, or configure policy sets using the administrative console.

Note: Policy sets do not support symmetric key encryption. If you are using the WSS API for symmetric key encryption, you will not be able to interoperate with web services endpoints that use policy sets.

If you are using the WSS APIs, the WSSDecryption and WSSDecryptPart APIs specify which algorithm methods are used when the client decrypts the SOAP messages.

- Use the WSSDecryption API to configure the data encryption algorithm and the key encryption algorithm methods.
- Use the WSSDecryptPart API to configure a transform algorithm method.

The WSS API process completes the following high-level steps to specify which decryption and decrypted part algorithm methods to use when configuring the client for response decryption:

Procedure

1. Using the WSSDecryption API, adds the required data encryption algorithm. The data encryption algorithm is used for encrypting or decrypting parts of a SOAP message. Data decryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method.

   The default data encryption algorithm is AES 128. The data encryption name is AES128, and the URI of the data encryption algorithm, is http://www.w3.org/2001/04/xmlenc#aes128-cbc. WebSphere Application Server supports the following pre-configured data decryption algorithms:

   - AES128: http://www.w3.org/2001/04/xmlenc#aes128-cbc

     The AES 128 algorithm is the default data algorithm method.
   - AES256: http://www.w3.org/2001/04/xmlenc#aes256-cbc

     To use this AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - AES192: http://www.w3.org/2001/04/xmlenc#aes192-cbc

     Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).

     To use this AES 192-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

2. As needed, changes the WSSEncryption API method to specify another data encryption algorithm. For example, you might add the following code to change from the default AES 128 algorithm to the Triple DES algorithm:

```
dec.addAllowedKeyEncryptionMethod(WSSDecryption.TRIPLE_DES);
```

3. Using the WSSDecryption API, adds the required key encryption algorithm. The key encryption algorithm is used for encrypting the key that is used for encrypting the message parts within the SOAP message. If no key for encrypting the data is needed, then you must specify WSSDecryption.encryptKey(false).

   The key encryption algorithm that you select for the consumer side must match the key encryption method that you select for the generator side.

   The default key encryption algorithm value is key wrap RSA_OAEP. The key encryption name is KW_RSA_OAEP, and the URI of the key encryption algorithm is http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p. WebSphere Application Server supports the following pre-configured key encryption algorithms:

   - KW_AES128: http://www.w3.org/2001/04/xmlenc#kw-aes128
   - KW_AES192: http://www.w3.org/2001/04/xmlenc#kw-aes192

     To use this key wrap AES 192 algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

     Restriction: Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
   - KW_AES256: http://www.w3.org/2001/04/xmlenc#kw-aes256

     To use this key wrap AES 256-cbc algorithm, you must download the unrestricted Java Cryptography Extension (JCE) policy file from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.
   - KW_RSA_OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p.

     The KW_RSA_OAEP algorithm is the default key algorithm method.

     When running with Software Development Kit (SDK) Version 1.4, the list of supported key transport algorithms does not include this algorithm. This algorithm appears in the list of supported key transport algorithms when running with SDK Version 1.5. See more information at http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
   - KW_RSA_15: http://www.w3.org/2001/04/xmlenc#rsa-1_5
   - KW_TRIPLE_DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

   Note: For Web Services Secure Conversation, the WSSEncryption API might specify addition key-related information, such as the:
   - algorithmName
   - keyLength

4. As needed, uses the WSSDecryption API method to change to other key encryption algorithms. For example, you might add the following code to change from the default key encryption algorithm KW_RSA_OAEP to the TRIPLE_DES algorithm:

```
dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);
```

5. Using the WSSDecryptPart API, adds a transform algorithm, as needed. There is no default transform algorithm. However, WebSphere Application Server provides a pre-configured decrypted part, `WSSDecryptPart.TRANSFORM_ATTACHMENT_CIPHERTEXT`, that can be added.

Results

If there is an error condition, a WSSException is provided. If successful, the API calls the `WSSConsumerContext.process()` method, the WS-Security header is validated, and the SOAP message is now secured using Web Services Security.

Example

The following example provides sample WSS API code for decrypting the body content as well as changing the data encryption and key encryption algorithms from the default values:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
                                 "",
                                 "enc-sender.jceks",
                                 "jceks",
                                 "storepass".toCharArray(),
                                 "alice",
                                 "keypass".toCharArray(),
                                 "CN=Alice, O=IBM, C=US");

// Generate WSSDecryption instance
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
                                                callbackHandler);

// Set the candidates for the data encryption method
// DEFAULT : WSSDecryption.AES128
   dec.addAllowedEncryptionMethod(WSSDecryption.AES128);
   dec.addAllowedEncryptionMethod(WSSDecryption.AES192);

// Set the candidates for the key encryption method
// DEFAULT : WSSDecryption.KW_RSA_OAEP
   dec.addAllowedKeyEncryptionMethod(WSSDecryption.KW_TRIPLE_DES);

// Add the WSSDecryption to WSSConsumingContext
   concont.add(dec);

// Validate the WS-Security header
concont.process(msgcontext);
```

Adding decrypted parts using the WSSDecryptPart API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To configure decrypted parts for the response consumer (client side) bindings, use the WSSDecryptPart API to define and add to the listing of elements in the decrypted part. WSSDecryptPart is an interface that is part of the com.ibm.websphere.wssecurity.wssapi.decryption package.

Before you begin

You can use either the WSS APIs or configure the policy sets using the administrative console to configure and add new encrypted parts. To secure SOAP messages using the WSSDecryptPart APIs, you must configure the decrypted parts for the response consumer bindings.

About this task

Confidentiality settings require that confidentiality constraints be applied to generated messages. These constraints include specifying which message parts within the generated message must be encrypted and decrypted, and which message parts to attach encrypted elements to.

The WSSDecryptPart API specifies information related to decryption and sets the decrypted parts that have been added for message confidentiality protection. Use the WSSDecryptPart to set the transform method and to specify the part to which the transform method is to be applied. Sets the transform method only if using SOAP with Attachments. The WSSDecryptPart is usually not needed except, in some case for tasks such as setting the transform method.

The decrypted parts displayed in the following table are used to protect the confidentiality of messages.

Table 213. Decrypted Parts. Use the decrypted parts to secure messages.

Decrypted parts	Description
keyword	Sets the decrypted part using keywords. The default decrypted parts that you can add using keywords are the BODY_CONTENT and SIGNATURE. WebSphere Application Server supports the following keywords: • BODY_CONTENT • SIGNATURE • USERNAME_TOKEN
xpath	Sets the decrypted part by using an XPath expression.
verification	Sets the WSSVerification component as a decrypted part. The WSSVerification part is applicable only if the SOAP message contains a signature element.
header	Sets the header, specified by QName, as a decrypted part.

For decrypted parts, certain default behaviors occur. The simplest way to use the WSSDecryptPart API is to use the default behavior (see the example code).

WSSDecryptPart provides defaults for setting the transform algorithm, adding a transform method, setting objects as targets, whether an element, and the encrypted parts, such as: the SOAP body content and the signature.

Table 214. Decrypted part decisions. Several characteristics of decrypted parts are configured by default.

Decryption decisions	Default behavior
Which SOAP message parts to decrypt using keywords	Specifies which keywords to use for the decrypted parts. WebSphere Application Server sets the following SOAP message parts by default for decryption: • WSSDecryption.BODY_CONTENT • WSSDecryption.SIGNATURE
Which transform algorithm to use (algorithm)	WebSphere Application Server does not specify any transform algorithm by default. Specify a transform method only if using SOAP with Attachments.

Procedure

1. To decrypt the SOAP message parts using the WSSDecryptPart API, first ensure that the application server is installed.
2. The WSS API process using WSSDecryptPart follows these steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates the SecurityToken from WSSFactory to configure decryption.
   d. Creates WSSDecryption from the WSSFactory instance using SecurityToken.
   e. Creates WSSDecryptPart from the WSSFactory instance. The default behavior of WSSDecryptPart is to assume that the body content and signature are encrypted.
   f. Adds the parts to be decrypted and to be applied with the transform in WSSDecryptPart. WebSphere Application Server sets these encrypted parts by default for WSSDecryptPart: the BODY_CONTENT and SIGNATURE. After you add other decrypted parts, the default values are no longer valid. For example, if you call addDecryptPart(securityToken, false), only the security token is encrypted, and not the signature and body content. So if you want to decrypt the security token, the signature, and the body content, you must call addDecryptPart(securityToken, false), addDecryptPart(WSSDecryption.SIGNATURE), and addDecryptPart(WSSDecryption.BODY_CONTENT).
   g. Sets the transform method.
   h. Adds WSSDecryptPart to WSSDecryption.
   i. Adds WSSDecryption to WSSConsumingContext.
   j. Calls WSSConsumingContext.process() with the SOAPMessageContext

Results

If there is an error condition when decrypting the message, a WSSException is provided. If successful, the API calls the WSSConsumingContext.process(), the WS-Security header is generated, and the SOAP message is now secured using Web Services Security.

What to do next

After enabling decrypted parts for the response consumer (client side) binding, specify the generator and consumer tokens, if the security tokens have not already been specified.

Decryption methods:

The decryption algorithms specify the data and key encryption algorithms that are used to decrypt the SOAP message. The WSS API for decryption (WSSDecryption) specifies the algorithm uniform resource identifier (URI) of the data and key encryption methods. The WSSDecryption interface is part of the `com.ibm.websphere.wssecurity.wssapi.decryption` package.

Data encryption algorithms

The data encryption algorithms are the algorithms that are used to encrypt and decrypt data. This algorithm type is used for encrypting data to encrypt and decrypt various parts of the message, including the body content and the signature.

Data decryption algorithms specify the algorithm uniform resource identifier (URI) of the data encryption method. WebSphere Application Server supports the following pre-configured data decryption algorithms:

Table 215. Supported pre-configured data decryption algorithms. The algorithms are used to decrypt SOAP messages.

WSS API	URI
WSSDecryption.AES128 (the default value)	A URI of data encryption algorithm, AES 128: http://www.w3.org/2001/04/xmlenc#aes128-cbc
WSSDecryption.AES192	A URI of data encryption algorithm, AES 192: http://www.w3.org/2001/04/xmlenc#aes192-cbc
WSSDecryption.AES256	A URI of data encryption algorithm, AES 256: http://www.w3.org/2001/04/xmlenc#aes256-cbc
WSSDecryption.TRIPLE_DES	A URI of data encryption algorithm, TRIPLE DES: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default, the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256-bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country, its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

For the AES256-cbc and the AES192-cbc algorithms, you must download the unrestricted Java™ Cryptography Extension (JCE) policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The data encryption algorithm must match the data decryption algorithm that is configured for the consumer.

Key encryption algorithms

The key encryption algorithms are the algorithms that are used to encrypt and decrypt keys.

This key information is used to specify the configuration that is needed to generate the key for digital signature and encryption. The signing information and encryption information configurations can share the key information. The key information on the consumer side is used for specifying the information about the key that is used for validating the digital signature in the received message or for decrypting the encrypted parts of the message. The request generator is configured for the client.

Key encryption algorithms specify the algorithm uniform resource identifier (URI) of the key encryption method. WebSphere Application Server supports the following pre-configured key encryption algorithms:

Table 216. Supported pre-configured key encryption algorithms. The algorithms are used to decrypt SOAP messages.

WSS API	URI
WSSDecryption.KW_AES128	A URI of key encryption algorithm, key wrap AES 128: http://www.w3.org/2001/04/xmlenc#kw-aes128
WSSDecryption.KW_AES192	A URI of key encryption algorithm, key wrap AES 192: http://www.w3.org/2001/04/xmlenc#kw-aes192 **Restriction:** Do not use the 192-bit key encryption algorithm if you want your configured application to be in compliance with the Basic Security Profile (BSP).
WSSDecryption.KW_AES256	A URI of key encryption algorithm, key wrap AES 256: http://www.w3.org/2001/04/xmlenc#kw-aes256
WSSDecryption.KW_RSA_OAEP (the default value)	A URI of key encryption algorithm, key wrap RSA OAEP: http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
WSSDecryption.KW_RSA15	A URI of key encryption algorithm, key wrap RSA 1.5: http://www.w3.org/2001/04/xmlenc#rsa-1_5
WSSDecryption.KW_TRIPLE_DES	A URI of data encryption algorithm, key wrap TRIPLE DES: http://www.w3.org/2001/04/xmlenc#kw-tripledes

By default, the RSA-OAEP algorithm uses the SHA1 message digest algorithm to compute a message digest as part of the encryption operation. Optionally, you can use the SHA256 or SHA512 message digest algorithm by specifying a key encryption algorithm property. The property name is: `com.ibm.wsspi.wssecurity.enc.rsaoaep.DigestMethod`. The property value is one of the following URIs of the digest method: `http://www.w3.org/2001/04/xmlenc#sha256` `http://www.w3.org/2001/04/xmlenc#sha512`

By default, the RSA-OAEP algorithm uses a null string for the optional encoding octet string for the OAEPParams. You can provide an explicit encoding octet string by specifying a key encryption algorithm property. For the property name, you can specify `com.ibm.wsspi.wssecurity.enc.rsaoaep.OAEPparams`. The property value is the base 64-encoded value of the octet string.

Important: You can set these digest method and OAEPParams properties on the generator side only. On the consumer side, these properties are read from the incoming SOAP message.

For the kw-aes256 and the kw-aes192 key encryption algorithms, you must download the unrestricted JCE policy files from the following website: http://www.ibm.com/developerworks/java/jdk/security/index.html.

The key encryption algorithm for the generator and the consumer must match.

The following example provides a sample of the WSS API code for the default algorithms that are used for WebSphere Application Server decryption:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();

// Required to attach username token into the message.
X509ConsumeCallbackHandler callbackHandler =
       new X509ConsumeCallbackHandler("",
                                      "enc-sender.jceks",
```

```
                                    "JCEKS",
                                    "storepass".toCharArray(),
                                    "alice",
                                    "keypass".toCharArray(),
                                    "CN=Alice, O=IBM, C=US");
// Set the decrypt component.
// Default encrypted part: Body-Content
// Default data encryption algorithm: AES128
// Default key encryption algorithm: KW-RSA-OAEP
WSSDecryption dec = factory.newWSSDecryption(X509Token.Type,
callbackHandler);
concont.add(dec);

// validate the WS-Security header.
concont.process(msgctx);
```

Verifying consumer signing information to protect message integrity using WSS APIs:

You can verify the signing information to protect message integrity for the response (client side) consumer binding. Signing information includes the signature and the signed parts for the generator side as well as signature verification and verify parts for the consumer side. To keep the integrity of the message, digital signatures are typically applied.

Before you begin

Ensure that the signature and signed parts information has been configured. The signature verification information must match what was configured on the generator side.

About this task

Integrity refers to digital signature while confidentiality refers to encryption. Integrity is provided by applying a digital signature to a SOAP message. To configure the signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity decreases the risk of data modification when you transmit data across a network.

Also, message integrity is provided by verifying the digitally signed body, time stamp, and WS-Addressing headers using the signature verification algorithm methods. The WSS APIs specify which algorithm is to be used to verify the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature verification method. WebSphere Application Server supports several pre-configured verification algorithm methods.

You can use the following interfaces to configure Web Services Security and to protect SOAP message integrity:

- Use the administrative console to configure policy sets for signature verification.
- Use the Web Services Security APIs (WSS API) to configure the SOAP message context (only for the client)

Perform the following verification tasks, using the WSS APIs, to configure the signing information and to protect message integrity for the consumer binding.

Procedure

- Configure the signing information using the WSSSignature API. Configure the signature verification information for the consumer binding using the WSSVerification API. Signature verification information is used to verify parts of a message including the SOAP body, the time stamp, and the WS-Addressing headers. Both verifying and decryption can be applied to the same message parts, such as the SOAP body.
- Add or change verify parts using the WSSVerifyPart API.

- Configure the client for request signing methods using the WSSVerification or WSSVerifyPart APIs. To configure the client for response verification, choose the verification methods. Use the WSSVerification API to configure the canonicalization and signature methods. Use the WSSVerifyPart API to configure the digest and transform methods.

Results

By completing the steps in these tasks, you have configured the consumer verification information to protect the integrity of messages.

Verifying signing information for the consumer binding using the WSS APIs:

You can configure the signing information for the client-side response consumer (receiver) bindings. Signing information is used to sign and validate parts of a message including the SOAP body, the timestamp information, and the Username token.

Before you begin

WebSphere Application Server uses XML digital signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method. Prior to completing these steps, read the information about XML digital signature to become familiar with signing and verifying digital signatures for digital content.

By including XML signature in SOAP messages, the following issues are realized: message integrity and authentication. *Integrity* refers to digital signature whereas confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet.

Before you can verify the signature and SOAP message signed parts, you must have completed the following tasks:

- Configured the signature.
- Added signed parts, as needed.
- Chosen the signature and signed parts methods.

About this task

Use the Web Services Security APIs (WSS API) to configure the signing verification information for the response consumer (client side) section of the bindings file. Use the WSSVerification or WSSVerifyPart APIs to configure the client for request signature verification and to specify which digitally signed message parts to verify.

WebSphere Application Server uses the signing information on the consumer side to verify the integrity of the received SOAP message by validating that the message parts (such as the body, time stamp, and Username token) are signed.

On the client side, use the WSS APIs, or configure policy sets using the administrative console to specify which parts of the message are signed and to configure the key information that is referenced by the key information references. To verify the signature and signed parts, use the WSSVerification and WSSVerifyPart APIs.

WebSphere Application Server provides default values for bindings. However, an administrator must modify the defaults for a production environment.

The WSSVerification and WSSVerifyPart APIs complete the following steps to specify which digitally signed message parts to verify when configuring the client for response consumer signing:

Procedure

1. The WSSVerification API adds the required verify parts of the SOAP message.

   The part reference refers to the message part that is digitally signed. The part attribute refers to the name of the <Integrity> element when the <PartReference> element is specified for the signature. You can specify multiple <PartReference> elements within the <SigningInfo> element. The <PartReference> element has two child elements when it is specified for the signature: <DigestTransform> and <Transform>.

   The WSSVerification API configures the following parts as verification parts:

Verification part	Description
Security token	Adds information for the security token that is used for the signature verification.
SOAP header and the QName as a target	Adds the SOAP header, specified by QName, as a verification part.

   The WSS APIs allow the use of keywords or an XPath expression to specify which parts of the message are to be verified. WebSphere Application Server supports the use of the following keywords:

Keyword	References
WSSVerification.ADDRESSING_HEADERS	The Web Services Addressing (WS-Addressing) headers.
WSSVerification.BODY	The SOAP message body. The body is the user data portion of the message.
WSSVerification.TIMESTAMP	The creation and expiration timestamp information.

2. The WSSVerification API adds the required header to the SOAP message. The header, specified by QName, is a required verification header.
3. The WSSVerification API adds a security token. Adds information about the security token that is to be used for the signature verification, such as:
   - The class for security token.
   - The callback handler
   - The name of the JAAS login configuration.
4. The WSSVerification API adds the signature method algorithm. The signature method is the algorithm that is used to convert the canonicalized <SignedInfo> element in the binding file into the <SignatureValue> element. The algorithm that is specified for the consumer, which is the response consumer configuration, must match the algorithm specified for the request generator configuration. WebSphere Application Server supports the following pre-configured signature algorithms:
   - WSSVerification.RSA_SHA1:http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSVerification.HMAC_SHA1:http://www.w3.org/2000/09/xmldsig#hmac-sha1

   WebSphere Application Server does not support the following algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1. You cannot use the DSA-SHA1 algorithm if you want to be compliant with the Basic Security Profile (BSP).
5. The WSSVerification API adds a canonicalization method. The canonicalization method algorithm is used to canonicalize the <SignedInfo> element before it is incorporated as part of the digital signature operation. The canonicalization algorithm that you specify for the generator must match the algorithm for the consumer.

   WebSphere Application Server supports the following pre-configured canonicalization algorithms:
   - WSSVerification.EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerification.C14N: http://www.w3.org/TR/xml-c14n
6. The WSSVerification API verifies whether a signature confirmation is required. The OASIS Web Services Security (WS-Security) Version 1.1 specification defines the use of signature confirmation. If you are using WS-Security Version 1.0, this function is not available.

The signature confirmation value is stored in order to validate the signature confirmation with it after the receiving message is returned. This method is called if the response message is expected to attach the signature confirmation into the SOAP message.

7. The WSSVerifyPart API adds a digest method. For each part reference in the signing information, the API specifies both a digest method algorithm and a transform algorithm.

   WebSphere Application Server supports the following pre-configured digest algorithms:

   - WSSVerifyPart.SHA1: http://www.w3.org/2000/09/xmldsig#sha1
   - WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512

8. The WSSVerifyPart API adds a transform method. For each part reference in the signing information, the API specifies both a digest method algorithm and a transform algorithm.

   WebSphere Application Server supports the following pre-configured transform algorithms:

   - WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support these algorithms:

   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/TR/1999/REC-xpath-19991116

   The transform algorithm for the consumer must match the transform algorithm for the generator.

Results

You have completed the steps to configure the signing information for the client-side response consumer sections of the bindings files.

Example

The following example shows WSS API sample code to verify the signature and to verify the X.509 token type as the security token:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();
// Generate the X.509 Callback Handler on the consumer side
    X509ConsumeCallbackHandler callbackhandler = generateCallbackHandler();
    WSSVerification ver = factory.newWSSVerification(X509Token.class,
        callbackhandler);
concont.add(ver);
```

What to do next

If not already configured, specify a similar signing information configuration for the generator bindings.

Next, if already configured, configure the encryption and decryption information, or configure the consumer and generator tokens.

Verifying the signature using the WSSVerification API:

You can secure the SOAP messages, without using policy sets for configuration, by using the Web Services Security APIs (WSS API). To verify the signing information for the consumer binding sections for the client side request, use the WSSVerification API. You must also specify which algorithm methods and which signature parts of the SOAP message are to be verified. The WSSVerification API is part of the com.ibm.websphere.wssecurity.wssapi.verification package.

Before you begin

Use the WSS APIs, or configure the policy sets by using the administrative console to verify the signing information. To secure SOAP messages, you must complete the following signature tasks:

- Configure the signature information.
- Choose the algorithm methods for signature and signature verification.
- Verify the signature information.

About this task

WebSphere Application Server uses the signing information for the default generator to sign parts of the message, and uses XML digital signature with existing algorithms such as RSA-SHA1 and HMAC-SHA1.

XML signature defines many methods for describing key information and enables the definition of a new method. XML canonicalization (C14N) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. The C14N process is used to canonicalize XML information. Select an appropriate C14N algorithm because the information that is canonicalized depends on this algorithm.

The following table shows the required and optional binding information when the digital signature security constraint (integrity) is defined.

Table 217. Signature verification parts. Use the signature verification parts to secure messages.

Verification parts	Description
keywords	Adds required signature parts as targets of verification by using keywords . Different message parts can be specified in the message protection for request on the generator side. Use the following keywords for the required signature verification parts: • ADDRESSING_HEADERS • BODY • TIMESTAMP The WS-Addressing headers are not encrypted but can be signed.
xpath	Adds verification parts by using an XPath expression.
part	Adds the WSSVerifyPart object as a verification part.
header	Adds the header, specified by QName, as a verification part.

For signature verification information, certain default behaviors occur. The simplest way to use the WSSVerification API is to use the default behavior.

The default values are defined by the WSS API for the digest method, the transform method, the security token, and the required verification parts.

Table 218. Signature verification default behaviors. Several characteristics of the signature verification parts are configured by default.

Signature verification decisions	Default behavior
Which signature method to use (algorithm)	Sets the signature algorithm method. Both the data encryption and the signature and the canonicalization can be specified. The default signature method is RSA SHA1. WebSphere Application Server supports the following pre-configured signature methods: • WSSVerification.RSA_SHA1: http://www.w3.org/2000/09/xmldsig#rsa-sha1 • WSSVerification.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1 The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.
Which canonicalization method to use (algorithm)	Sets the canonicalization algorithm method. Both the data encryption and the signature and the canonicalization can be specified. The default signature method is EXC_C14N. WebSphere Application Server supports the following pre-configured canonicalization methods: • WSSVerification.EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n# • WSSVerification.C14N: http://www.w3.org/2001/10/xml-c14n#

Table 218. Signature verification default behaviors (continued). Several characteristics of the signature verification parts are configured by default.

Signature verification decisions	Default behavior
Whether signature confirmation is required	If the WSSSignature API specifies that signature confirmation is required, then the WSSVerification API verifies the signature confirmation value in the response message that has the signature confirmation value attached to it when received. Signature confirmation is defined in the OASIS Web Services Security Version 1.1 specification. The default signature confirmation is false.
Which security token to specify (securityToken)	Adds the securityToken object as a signature part. WebSphere Application Server sets the token information to use for verification. WebSphere Application Server supports the following pre-configured tokens for signing: • X.509 Token • Derived Key Token Information required for tokens include the class for the token, the callback handler information, and the name of the JAAS login module.

Procedure

1. To verify the signature in a SOAP message by using the WSSVerification API, first ensure that the application server is installed.
2. Use the WSSVerification API to set the message parts to be verified and to specify the algorithms in a SOAP message. The WSS API process for signature verification follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance.
   c. Ensures that WSSConsumingContext is called in the JAX-WS Provider implementation class. Due to the nature of the JAX-WS programming model, a JAX-WS provider needs to be implemented and must call the WSSConsumingContext to verify the SOAP message signature.
   d. Creates WSSVerification from the WSSFactory instance.
   e. Adds the part to be verified. If the digest method or the transform method are changed, create WSSVerifyPart and set it into WSSVerification.
   f. Sets the candidates of the canonicalization method, if the default is not appropriate.
   g. Sets the candidates of the signature method, if the default is not appropriate.
   h. Sets the candidate security token, if the default is not appropriate.
   i. Calls the requireSignatureConfirmation(), if the signature confirmation is applied.
   j. Adds WSSVerification to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAP message context.

Results

You have completed the steps to verify the signature for the consumer section of the bindings. If there is an error condition, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is applied to the SOAP message.

Example

The following example provides sample code that uses methods that are defined in the WSSVerification API:

```
// Get the message context
  Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext concont = factory.newWSSConsumingContext();
```

```
// Generate the certificate list
  String certpath = "c:/WebSphere/AppServer/etc/ws-security/samples/intca2.cer";
// The location of the X509 certificate file
  X509Certificate x509cert = null;
     try {
      InputStream is = new FileInputStream(certpath);
      CertificateFactory cf = CertificateFactory.getInstance("X.509");
      x509cert = (X509Certificate)cf.generateCertificate(is);
       } catch(FileNotFoundException e1){
        throw new WSSException(e1);
       } catch (CertificateException e2) {
        throw new WSSException(e2);
       }
      Set<Object> eeCerts = new HashSet<Object>();
       eeCerts.add(x509cert);
// Create the certificate store
   java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
   CollectionCertStoreParameters certparam = new CollectionCertStoreParameters(eeCerts);
   CertStore cert = null;
   try {
    cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
   } catch (NoSuchProviderException e1) {
    throw new WSSException(e1);
   } catch (InvalidAlgorithmParameterException e2) {
    throw new WSSException(e2);
   } catch (NoSuchAlgorithmException e3) {
    throw new WSSException (e3);
   }
   if(certList != null ){
    certList.add(cert);
   }
// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new X509ConsumeCallbackHandler(
     "dsig-receiver.ks",
     "jks",
     "server".toCharArray(),
     certList,
     java.security.Security.getProvider("IBMCertPath")
   );

// Generate the WSSVerification instance (step: d)
  WSSVerification ver = factory.newWSSVerification(X509Token.class, callbackHandler);

// Set the part to be verified (step: e)
// DEFAULT: WSSVerification.BODY, WSSSignature.ADDRESSING_HEADERS,
// and WSSSignature.TIMESTAMP.

// Set the part in the SOAP header to be specified by QName (step: e)
     ver.addRequiredVerifyHeader(new QName("http://www.w3.org/2005/08/addressing", "MessageID"));

// Set the part to be specified by the keyword (step: e)
     ver.addRequiredVerifyPart(WSSVerification.BODY);

// Set the part to be specified by WSSVerifyPart (step: e)
  WSSVerifyPart verPart = factory.newWSSVerifyPart();
     verPart.setRequiredVerifyPart(WSSVerification.BODY);
     verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);
     ver.addRequiredVerifyPart(verPart);

// Set the part specified by XPath expression (step: e)
  StringBuffer sb = new StringBuffer();
        sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Envelope']");
        sb.append("/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/'
            and local-name()='Body']");
        sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Ping']");
        sb.append("/*[namespace-uri()='http://xmlsoap.org/Ping'
            and local-name()='Text']");
     ver.addRequiredVerifyPartByXPath(sb.toString());


// Set one or more canonicalization method candidates for verification (step: f)
// DEFAULT : WSSVerification.EXC_C14N
    ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
    ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Set one or more signature method candidates for verification (step: g)
// DEFAULT : WSSVerification.RSA_SHA1
    ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

// Set the candidate security token to used for the verification (step: h)
  X509ConsumeCallbackHandler callbackHandler2 = getCallbackHandler2();
     ver.addToken(X509Token.class, callbackHandler2);


// Set the flag to require the signature confirmation (step: i)
   ver.requireSignatureConfirmation();
```

```
// Add the WSSVerification to the WSSConsumingContext (step: j)
  concont.add(ver);

//Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

What to do next

After verifying the signature and setting algorithm methods for the SOAP message, you can set either the digest method or the transform method. If you want to set these methods, use the WSSVerifyPart API, or configure policy sets using the administrative console.

Verifying signed parts using the WSSVerifyPart API:

To secure SOAP messages on the consumer side, use the Web Services Security APIs (WSS API) to configure the verify parts information for the consumer binding on the response consumer (client side). You can specify which algorithm methods and which parts of the SOAP message are to be verified. Use the WSSVerifyPart API to change the digest method or the transform method. The WSSVerifyPart API is part of the com.ibm.websphere.wssecurity.wssapi.verification package.

Before you begin

To secure SOAP messages using the signing verification information, you must complete one of the following tasks:

- Configure the signature verification information using the WSSVerification API.
- Configure verify parts using the WSSVerifyPart API, as needed.

The WSSVerifyPart is used for specify the transform or digest methods for the verification. Use the WSSVerifyPart API or configure policy sets using the administrative console.

About this task

WebSphere Application Server uses the signing information for the default consumer to verify the signed parts of the message. The WSSVerifyPart API is only supported on the response consumer (requester).

The following table shows the required verification parts when the digital signature security constraint (integrity) is defined:

Table 219. Verify parts information. Use the verify parts to secure messages with signing verification information.

Verify parts information	Description
keyword	Sets the verify parts using the following keywords: • BODY • ADDRESSING_HEADERS • TIMESTAMP The WS-Addressing headers are not decrypted but can be signed and verified.
xpath	Sets the verify parts using an XPath expression.
header	Sets the header, specified by QName, as a required verify part.

For signature verification, certain default behaviors occur. The simplest way to use the WSSVerification API is to use the default behavior (see the example code). The default values are defined by the WSS API for the signing algorithm and the canonicalization algorithm, and the verify parts.

Table 220. Verify parts default behaviors. Several characteristics of verify parts are configured by default.

Verify parts decisions	Default behavior
Which keywords to specify	The different SOAP message parts to be signed and used for message protection. WebSphere Application Server supports the following keywords: • WSSVerification.BODY • WSSVerification.ADDRESSING_HEADERS • WSSVerification.TIMESTAMP
Which transform method to use (algorithm)	Adds the transform method. The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signature. The default transform method is TRANSFORM_EXC_C14N. WebSphere Application Server supports the following pre-configured transform algorithms: • WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n# • WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2 Use this transform method to ensure compliance with the Basic Security Profile (BSP). • WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform • WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature
Which digest method to use (algorithm)	Sets the digest algorithm method. The digest method algorithm that is specified within the <DigestMethod> element is used in the <SigningInfo> element. The default digest method is SHA1. WebSphere Application Server supports the following digest method algorithms: • WSSVerifyPart.SHA1: http://www.w3.org/2000/09/xmldsig#sha1 • WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256 • WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512

Procedure

1. To verify signed parts by using the WSSVerifyPart API, first ensure that the application server is installed.
2. Use the Web Services Security API to verify the verification in a SOAP message. The WSS API process for verifying the signature follows these process steps:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Ensures that WSSConsumingContext is called in the JAX-WS Provider implementation class. Due to the nature of the JAX-WS programming model, a JAX-WS provider needs to be implemented and must call the WSSConsumingContext to verify the SOAP message signature.
   c. Creates the CallbackHandler to use for verification.
   d. Create the WSSVerification object from the WSSFactory instance.
   e. Creates WSSVerifyPart from the WSSFactory instance.
   f. Sets the part to be verified, if the default is not appropriate.
   g. Sets the candidates for the digest method, if the default is not appropriate.
   h. Sets the candidates for the transform method, if the default is not appropriate.
   i. Adds WSSVerifyPart to WSSVerification.
   j. Adds WSSVerification to WSSConsumingContext.
   k. Calls WSSConsumingContext.process() with the SOAPMessageContext.

Results

You have completed the steps to verify to verify the signed parts on the consumer side. If there is an error condition when verifying the signing information, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and Web Services Security is verified for the SOAP message.

Example

The following example provides sample code for the WSSVerification API process for verifying the signing information in a SOAP message:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext concont = factory.newWSSConsumingContext();

// Generate the certificate list
   String certpath =
   "c:/WebSphere/AppServer/etc/ws-security/samples/intca2.cer";
// The location of the X509 certificate file
   X509Certificate x509cert = null;
      try {
          InputStream is = new FileInputStream(certpath);
          CertificateFactory cf = CertificateFactory.getInstance("X.509");
          x509cert = (X509Certificate)cf.generateCertificate(is);
      } catch(FileNotFoundException e1){
            throw new WSSException(e1);
      } catch (CertificateException e2) {
            throw new WSSException(e2);
      }

      Set<Object> eeCerts = new HashSet<Object>();
      eeCerts.add(x509cert);
// create certStore
      java.util.List<CertStore> certList = new
          java.util.ArrayList<CertStore>();
      CollectionCertStoreParameters certparam = new
          CollectionCertStoreParameters(eeCerts);
      CertStore cert = null;
      try {
         cert = CertStore.getInstance("Collection",
              certparam, "IBMCertPath");
      } catch (NoSuchProviderException e1) {
           throw new WSSException(e1);
      } catch (InvalidAlgorithmParameterException e2) {
           throw new WSSException(e2);
      } catch (NoSuchAlgorithmException e3) {
           throw new WSSException (e3);
      }
      if(certList != null ){
      certList.add(cert);
      }

// generate callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
       X509ConsumeCallbackHandler(
                                "dsig-receiver.ks",
                                "jks",
                                "server".toCharArray(),
                                certList,
       java.security.Security.getProvider("IBMCertPath")
);

// Generate the WSSVerification instance (step: d)
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
       callbackHandler);

// Set the part to be specified by WSSVerifyPart (step: e)
   WSSVerifyPart verPart = factory.newWSSVerifyPart();

// Set the part to be specified by the keyword (step: f)
   verPart.setRequiredVerifyPart(WSSVerification.BODY);

// Set the candidates for the digest method for verification (step: g)
// DEFAULT : WSSVerifyPart.SHA1
   verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

// Set the candidates for the transform method for verification (step: h)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N : String
   verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

// Set WSSVerifyPart to WSSVerification (step: i)
   ver.addRequiredVerifyPart(verPart);

// Add WSSVerification to WSSConsumingContext (step: j)
   concont.add(ver);

//Validate the WS-Security header (step: k)
concont.process(msgcontext);
```

What to do next

You have completed configuring the signed part to be verified.

Configuring response signature verification methods for the client:

Use the WSSVerification and WSSVerifyPart APIs to choose the signing verification methods. The request signing verification methods include the digest algorithm and the transport methods.

Before you begin

To complete configuration of the signature verification information to secure SOAP messages, you must perform the following algorithm tasks:

- Use the WSSVerification API to configure the canonicalization and signature methods.
- Use the WSSVerifyPart API to configure the digest and transform methods.

to configure the algorithm methods to use when configuring the client for request signing.

About this task

The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 221. Signing verification methods. Use signing verification information to secure messages.

Name of method	Purpose
Digest algorithm	Applies to the data after transforms are applied, if specified, to yield the <DigestValue> element. Signing the <DigestValue> element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the client request receiver configuration.
Transform algorithm	Applies to the <Transform> element.
Signature algorithm	Specifies the Uniform Resource Identifiers (URI) of the signature verification method.
Canonicalization algorithm	Specifies the Uniform Resource Identifiers (URI) of the canonicalization method.

After configuring the client to digitally sign the message, you must configure the client to verify the digital signature. You can use the WSS APIs or configure policy sets using the administrative console to verify the digital signature and to choose the verification and verify part algorithms. If using the WSS APIs to configure, use the WSSVerification and WSSVerifyPart APIs to specify which digitally signed message parts to verify and to specify which algorithm methods to use when configuring the client for request signing.

The WSSVerification and WSSVerifyPart APIs perform the following steps to configure the signature verification and verify parts algorithm methods:

Procedure

1. For the consumer binding, the WSSVerification API specifies the signature methods to allow for the signature verification. WebSphere Application Server supports the following pre-configured signature methods:
   - WSSVerification.RSA_SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#rsa-sha1
   - WSSVerification.HMAC_SHA1: http://www.w3.org/2000/09/xmldsig#hmac-sha1

   The DSA-SHA1 digital signature method (http://www.w3.org/2000/09/xmldsig#dsa-sha1) is not supported.

2. For the consumer binding, the WSSVerification API specifies the canonicalization method to allow for the signature verification. WebSphere Application Server supports the following pre-configured canonicalization methods by default:
   - WSSVerification.EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerification.C14N: http://www.w3.org/2001/10/xml-c14n#
3. For the consumer binding, the WSSVerifyPart API specifies the digest method, as needed. WebSphere Application Server supports the following digest method algorithms for signed parts verification:
   - WSSVerifyPart.SHA1 (the default value): http://www.w3.org/2000/09/xmldsig#sha1
   - WSSVerifyPart.SHA256: http://www.w3.org/2001/04/xmlenc#sha256
   - WSSVerifyPart.SHA512: http://www.w3.org/2001/04/xmlenc#sha512
4. For the consumer binding, the WSSVerifyPart API specifies the transform method. WebSphere Application Server supports the following transform algorithms for verify parts:
   - WSSVerifyPart.TRANSFORM_EXC_C14N (the default value): http://www.w3.org/2001/10/xml-exc-c14n#
   - WSSVerifyPart.TRANSFORM_XPATH2_FILTER: http://www.w3.org/2002/06/xmldsig-filter2
   - WSSVerifyPart.TRANSFORM_STRT10: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
   - WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE: http://www.w3.org/2000/09/xmldsig#enveloped-signature

   For the WSS APIs, WebSphere Application Server does not support these algorithms:
   - http://www.w3.org/2002/07/decrypt#XML
   - http://www.w3.org/TR/1999/REC-xpath-19991116

Results

You have specified which method to use when verifying a digital signature when the client sends a message.

Example

The following example provides sample WSS API code that specifies the verification information, the body as a part to be verified, the HMAC_SHA1 as a signature method, C14N and EXC_C14N as the candidates of canonicalization methods, TRANSFORM_STRT10 as a transform method, and SHA256 as a digest method.

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance
   WSSConsumingContext concont = factory.newWSSConsumingContext();

// Generate the certificate list
   String certpath = "intca2.cer";
// The location of the X509 certificate file
      X509Certificate x509cert = null;
      try {
            InputStream is = new FileInputStream(certpath);
            CertificateFactory cf = CertificateFactory.getInstance("X.509");
           x509cert = (X509Certificate)cf.generateCertificate(is);
      } catch(FileNotFoundException e1){
          throw new WSSException(e1);
      } catch (CertificateException e2) {
          throw new WSSException(e2);
      }

      Set<Object> eeCerts = new HashSet<Object>();
      eeCerts.add(x509cert);
// Create the certStore
   java.util.List<CertStore> certList = new
        java.util.ArrayList<CertStore>();
   CollectionCertStoreParameters certparam = new
```

```
        CollectionCertStoreParameters(eeCerts);
   CertStore cert = null;
      try {
            cert = CertStore.getInstance("Collection",
                                         certparam,
                                        "IBMCertPath");
      } catch (NoSuchProviderException e1) {
          throw new WSSException(e1);
      } catch (InvalidAlgorithmParameterException e2) {
          throw new WSSException(e2);
      } catch (NoSuchAlgorithmException e3) {
          throw new WSSException (e3);
      }
      if(certList != null ){
           certList.add(cert);
      }

// Generate the callback handler
   X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
                                "dsig-receiver.ks",
                                "jks",
                                "server".toCharArray(),
                                certList,
                                java.security.Security.getProvider(
                                      "IBMCertPath")
                                );

// Generate the WSSVerification instance
   WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                    callbackHandler);

// Set one or more candidates of the signature method used for
//    verification (step. 1)
// DEFAULT : WSSVerification.RSA_SHA1
      ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

// Set one or more candidates of the canonicalization method used for
//    verification (step. 2)
// DEFAULT : WSSVerification.EXC_C14N
      ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
      ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

// Set the part to be specified by WSSVerifyPart
   WSSVerifyPart verPart = factory.newWSSVerifyPart();

// Set the part to be specified by the keyword
      verPart.setRequiredVerifyPart(WSSVerification.BODY);

// Set the candidates of digest methods to use for verification (step. 3)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N : String
      verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

// Set the candidates of digest methods to use for verification (step. 4)
// DEFAULT : WSSVerifyPart.SHA1
      verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

// Set WSSVerifyPart to WSSVerification
      ver.addRequiredVerifyPart(verPart);

// Add the WSSVerification to the WSSConsumingContext
   concont.add(ver);

// Validate the WS-Security header
concont.process(msgcontext);
```

What to do next

You have completed configuring the signature verification algorithms. Next, configure the encryption or decryption algorithms, if not already configured. Or, configure the security token information, as needed.

Signature verification methods using the WSSVerification API:

You can verify the signing or signature information using the WSS API for the consumer binding. The signature and canonicalization algorithm methods are used for the generator binding. The WSSVerification API is provided in the com.ibm.websphere.wssecurity.wssapi.verification package.

To configure consumer signing information to protect message integrity, you must first digitally sign and then verify the signature for the SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signature verification include the:

Signature method
Sets the signature algorithm method.

Canonicalization method
Sets the canonicalization algorithm method.

The algorithm that is specified for the request generator configuration must match the algorithm that is specified for the response consumer configuration.

Signature algorithms

The signature algorithms specify the signature verification algorithm that is used to sign the certificate. The signature algorithms specify the Uniform Resource Identifiers (URI) of the signature verification method. WebSphere Application Server supports the following pre-configured algorithms:

Table 222. Signature verification algorithms. The algorithms include the signature methods.

Algorithm	Description
WSSVerification.HMAC_SHA1	A URI of the signature algorithm, HMAC: http://www.w3.org/2000/09/xmldsig#hmac-sha1
WSSVerification.RSA_SHA1 (the default value)	A URI of the signature algorithm, RSA: http://www.w3.org/2000/09/xmldsig#rsa-sha1

WebSphere Application Server does not support the algorithm for DSA-SHA1: http://www.w3.org/2000/09/xmldsig#dsa-sha1

Canonicalization algorithms

The canonicalization algorithms specify the Uniform Resource Identifiers (URI) of the canonicalization method. WebSphere Application Server supports the following pre-configured algorithms:

Table 223. Verification canonicalization algorithms. The algorithms include the canonicalization methods.

Algorithm	Description
WSSVerification.C14N	A URI of the inclusive canonicalization algorithm, C14N: http://www.w3.org/2001/10/xml-c14n#
WSSVerification.EXC_C14N (the default value)	A URI of the exclusive canonicalization algorithm EXC_C14N: http://www.w3.org/2001/10/xml-exc-c14n#

The following example provides sample WSS API code that specifies the X.509 token security token for signature verification:

```
WSSFactory factory = WSSFactory.getInstance();
WSSConsumingContext concont = factory.newWSSConsumingContext();

// X509ConsumeCallbackHandler
X509ConsumeCallbackHandler callbackHandler = new
   X509ConsumeCallbackHandler("dsig-receiver.ks",
                              "jks",
                              "server".toCharArray(),
                              certList,
   java.security.Security.getProvider("IBMCertPath")46 );

// Set the verification component
// DEFAULT verification parts: Body, WS-Addressing header, and Timestamp
// DEFAULT data encryption algorithm: RSA-SHA1
// DEFAULT digest algorithm: SHA1
// DEFAULT canonicalization algorithm: exc-c14n
WSSVerification ver = factory.newWSSVerification(X509Token.class,
                                                 callbackhandler);
concont.add(ver);

// Validate the WS-Security header
concont.validate(msgctx);
```

Choosing the verify parts methods using the WSSVerifyPart API:

You can configure the signing verification information for the consumer binding using the WSS API. The transform algorithm and digest methods are used for the consumer binding. Use the WSSVerifyPart API to configure the algorithm methods. The WSSVerifyPart API is provided in the com.ibm.websphere.wssecurity.wssapi.verification package.

To configure consumer verify parts information to protect message integrity, you must first digitally sign and then verify the signature and signed parts for the SOAP messages. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification when you transmit data across a network.

Methods

Methods that are used for the signing information include the:

Digest method
Sets the digest method.

Transform method
Sets the transform algorithm method.

Digest algorithms

The digest method algorithm is specified within the element is used in the <Digest> element. WebSphere Application Server supports the following pre-configured digest algorithms:

Table 224. Verify parts digest methods. Use the verify parts to protect message integrity.

Digest method	Description
WSSVerifyPart.SHA1 (the default value)	A URI of the digest algorithm, SHA1: http://www.w3.org/2000/09/xmldsig#sha1
WSSVerifyPart.SHA256	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha256
WSSVerifyPart.SHA512	A URI of the digest algorithm, SHA256: http://www.w3.org/2001/04/xmlenc#sha512

Transform algorithms

The transform algorithm is specified within the <Transform> element and specifies the transform algorithm for the signed part. WebSphere Application Server supports the following pre-configured transform algorithms:

Table 225. Verify parts transform methods. Use the verify parts to protect message integrity.

Digest method	Description
WSSVerifyPart.TRANSFORM_ENVELOPED_SIGNATURE	A URI of the transform algorithm, enveloped signature: http://www.w3.org/2000/09/xmldsig#enveloped-signature
WSSVerifyPart.TRANSFORM_STRT10	A URI of the transform algorithm, STR-Transform: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform
WSSVerifyPart.TRANSFORM_EXC_C14N (the default value)	A URI of the transform algorithm, Exc-C14N: http://www.w3.org/2001/10/xml-exc-c14n#
WSSVerifyPart.TRANSFORM_XPATH2_FILTER	A URI of the transform algorithm, XPath2 filter: http://www.w3.org/2002/06/xmldsig-filter2

For the WSS APIs, WebSphere Application Server does not support the following transform algorithms:

- http://www.w3.org/TR/1999/REC-xpath-19991116
- http://www.w3.org/2002/07/decrypt#XML

The following example provides sample WSS API code that verifies the body using SHA256 as the digest method and TRANSFORM_EXC_14N and TRANSFORM_STRT10 as the transform methods:

```
// get the message context
Object msgcontext = getMessageContext();

// generate WSSFactory instance
WSSFactory factory = WSSFactory.getInstance();

// generate WSSConsumingContext instance
WSSConsumingContext concont = factory.newWSSConsumingContext();


// generate the cert list
String certpath = "intca2.cer";// The location of the X509
   certificate file X509Certificate x509cert = null;
try {
 InputStream is = new FileInputStream(certpath);
 CertificateFactory cf = CertificateFactory.getInstance("X.509");
 x509cert = (X509Certificate)cf.generateCertificate(is);
} catch(FileNotFoundException e1){
 throw new WSSException(e1);
} catch (CertificateException e2) {
 throw new WSSException(e2);
}

Set<Object> eeCerts = new HashSet<Object>();
eeCerts.add(x509cert);
// create certStore
java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
CollectionCertStoreParameters certparam = new
      CollectionCertStoreParameters(eeCerts);
CertStore cert = null;
try {
 cert = CertStore.getInstance("Collection", certparam, "IBMCertPath");
} catch (NoSuchProviderException e1) {
 throw new WSSException(e1);
} catch (InvalidAlgorithmParameterException e2) {
 throw new WSSException(e2);
} catch (NoSuchAlgorithmException e3) {
 throw new WSSException (e3);
}
if(certList != null ){
 certList.add(cert);
}

// generate callback handler
X509ConsumeCallbackHandler callbackHandler = new
      X509ConsumeCallbackHandler(
  "dsig-receiver.ks",
  "jks",
  "server".toCharArray(),
  certList,
  java.security.Security.getProvider("IBMCertPath")
);

//generate WSSVerification instance
WSSVerification ver = factory.newWSSVerification(X509Token.class,
     callbackHandler);

//set one or more candidates of the signature method used for the
 //verification (step. 1)
// DEFAULT : WSSVerification.RSA_SHA1
ver.addAllowedSignatureMethod(WSSVerification.HMAC_SHA1);

//set one or more candidates of the canonicalization method used
 //for the verification (step. 2)
// DEFAULT : WSSVerification.EXC_C14N
ver.addAllowedCanonicalizationMethod(WSSVerification.C14N);
ver.addAllowedCanonicalizationMethod(WSSVerification.EXC_C14N);

//set the part to be specified by WSSVerifyPart
WSSVerifyPart verPart = factory.newWSSVerifyPart();

//set the part to be specified by the keyword
verPart.setRequiredVerifyPart(WSSVerification.BODY);

//set the candidates of digest methods to use for verification (step. 3)
// DEFAULT : WSSVerifypart.TRANSFORM_EXC_C14N
verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_EXC_C14N);
verPart.addAllowedTransform(WSSVerifyPart.TRANSFORM_STRT10);

//set the candidates of digest methods to use for verification (step. 4)
// DEFAULT : WSSVerifyPart.SHA1
verPart.addAllowedDigestMethod(WSSVerifyPart.SHA256);

//set WSSVerifyPart to WSSVerification
ver.addRequiredVerifyPart(verPart);
```

```
//add the WSSVerification to the WSSConsumingContext
concont.add(ver);

//validate the WS-Security header
concont.process(msgcontext);
```

Validating the consumer token to protect message authenticity:

The token consumer information is used on the consumer side to incorporate and validate the security token. The Username token, X509 tokens, and LTPA tokens by default are used for message authenticity.

Before you begin

The token processing and pluggable token architecture in the Web Services Security run time reuses the same security token interface and Java Authentication and Authorization Service (JAAS) Login Module from the Web Services Security APIs (WSS API). The same implementation of token creation and validation can be used in both the WSS API and the WSS SPI in the Web Services Security run time.

Restriction: The `com.ibm.wsspi.wssecurity.token.TokenConsumingComponent` interface is not used with JAX-WS web services. If you are using JAX-RPC web services, this interface is still valid.

Note that the key name (KeyName) element is not supported because there is no KeyName policy assertion defined in the current OASIS Web Services Security draft specification.

About this task

The JAAS callback handler (CallbackHandler) and the JAAS login module (LoginModule) are responsible for creating the security token on the generator side and validating (authenticating) the security token on the consumer side.

For example, on the generator side, the Username token is created by the JAAS LoginModule and using the JAAS CallbackHandler to pass the authentication data. The JAAS LoginModule creates the Username SecurityToken object and passes it to the Web Services Security run time.

Then, on the consumer side, the Username Token XML format is passed to the JAAS LoginModule for validation or authentication and the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated, a Username SecurityToken object is created and passed it to the Web Services Security run time.

Note: WebSphere Application Server does not support a stackable login module with the WebSphere Application Server default login module implementation, meaning adding the login module before or after the WebSphere Application Server login module implementation. If you want to stack the login module implementations, you must develop the required login modules because there is no default implementation.

The com.ibm.websphere.wssecurity.wssapi.token package provided by WebSphere Application Server includes support for these classes:

- Security token (SecurityTokenImpl)
- Binary security token (BinarySecurityTokenImpl)

In addition, WebSphere Application Server provides the following pre-configured sub-interfaces for security tokens:

- Derived key token
- Security context token (SCT)
- Username token
- LTPA token propagation

- LTPA token
- X509PKCS7 token
- X509PKIPath token
- X509v3 token
- Kerberos v5 token

The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity. The derived key token and the X.509 tokens are used by default for signing and encryption.

The WSS API and WSS SPI are only supported on the client. To specify the security token type on the consumer side, you can also configure policy sets using the administrative console. You can also use the WSS APIs or policy sets for matching generator security tokens.

The default Login Module and Callback implementations are designed to be used as a pair, meaning both a generator and a consumer part. To use the default implementations, select the appropriate generator and consumer security token in a pair. For example, select `system.wss.generate.x509` in the token generator and `system.wss.consume.x509` in the token consumer when the X.509 token is required.

To configure the consumer-side security token, use the appropriate pre-configured token consumer interface from the WSS APIs to complete the following token configuration process steps:

Procedure

1. Generate the wssFactory instance.
2. Generate the wssConsumingContext instance.

   The WSSConsumingContext interface stores the components for consuming Web Services Security (WS-Security), such as verification, decryption, the security token, and the time stamp. When the validate() method is called, all of these components are validated.
3. Create the consumer-side components, such as the WSSVerification and the WSSDecryption objects.
4. Specify a JAAS configuration by specifying the name of the JAAS login configuration. The Java Authentication and Authorization Service (JAAS) configuration specifies the name of the JAAS configuration. The JAAS configuration specifies how the token logs in on the consumer side. Do not remove the predefined system or application login configurations. However, within these configurations, you can add module class names and specify the order in which WebSphere Application Server loads each module.
5. Specify a token consumer class name. The token consumer class name specifies the required information to validate the SecurityToken. The Username token, the X.509 tokens, and the LTPA tokens are used by default for message authenticity.
6. Specify the settings for the callback handler by specifying a callback handler class name and also specifies the callback handler keys. This class name is the name of the callback handler implementation class that is used for the plug-in to the security token framework.

   WebSphere Application Server provides the following default callback handler implementations for the consumer side:

   com.ibm.websphere.wssecurity.callbackhandler.PropertyCallback
   This class is a callback for handling the name-value pair in elements in the Web Services Security (WS-Security) configuration XMI files.

   ccom.ibm.websphere.wssecurity.callbackhandler.UNTConsumeCallbackHandler
   This class is a callback handler for the Username token on the consumer side. This instance is used to set into WSSConsumingContext object to validate a Username token. Use this implementation for a Java Platform, Enterprise Edition (Java EE) application client only.

   com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler
   This class is a callback handler that is used to validate the X.509 certificate that is inserted in

the Web Services Security header within the SOAP message as a binary security token on the consumer side. This instance is used to generate the WSSVerification object and WSSDecryption objects, set the objects into WSSConsumingContext object to validate the X.509 binary security tokens. A keystore and a key definition are required for this callback handler. If you use this implementation, a key store password, path, and type must have been provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.LTPAConsumeCallbackHandler
This class is a callback handler for the Lightweight Third Party Authentication (LTPA) tokens on the consumer side. This instance is used to generate the WSSVerification and WSSDecryption objects to validate an LTPA token.

This callback handler is used to validate the LTPA security token inserted in the Web Services Security header within the SOAP message as a binary security token. However, if the user name and password are specified, WebSphere Application Server authenticates the user name and password to obtain the LTPA security token rather than obtaining it from the Run As Subject. Use this callback handler only when the web service is acting as a client on the application server. It is recommended that you do not use this callback handler on a Java EE application client. If you use this implementation, a basic authentication user ID and password must have been provided on the generator side.

com.ibm.websphere.wssecurity.callbackhandler.KRBTokenConsumeCallbackHandler
This class is a callback handler for the Kerberos v5 token on the consumer side. This instance is used to set the WSSConsumingContext object to consume the Kerberos v5 AP-REQ as a binary security token. The instance is also used to generate the WSSVerification and WSSDecryption objects to use the Kerberos session key or derived key in the SOAP message verification and decryption.

7. If a X.509 token is specified, additional token information is also specified.

Table 226. Information for the X.509 token. Use the X.509 token to authenticate messages.

Token Information	Description
keyStoreRef	The reference name of the keystore that is used for the key locator.
keyStorePath	The keystore file path from which the keystore is loaded, if needed. It is recommended that you use the `${USER_INSTALL_ROOT}` in the path name as this variable expands to the WebSphere Application Server path on your machine. This path is required when you use the X.509 tokens callback handler implementations.
keyStorePassword	The password that is used to check the integrity of the keystore, or the keystore password that is used to unlock the keystore and to access the keystore file. The keystore and its configuration are used for some of the default callback handler implementations that are provided by WebSphere Application Server.
keyStoreType	The keystore type of keystore that is used for the key locator. This selection indicates the format that is used by the keystore file. The following values are available for selection: **JKS** Use this option if the keystore uses the Java Keystore (JKS) format. **JCEKS** Use this option if the Java Cryptography Extension is configured in the software development kit (SDK). The default IBM JCE is configured in WebSphere Application Server. This option provides stronger protection for stored private keys by using Triple DES encryption. **JCERACFKS** Use JCERACFKS if the certificates are stored in a SAF key ring (z/OS only). **PKCS11KS (PKCS11)** Use this format if your keystore uses the PKCS#11 file format. Keystores using this format might contain RSA keys on cryptographic hardware or might contain encrypt keys that use cryptographic hardware to ensure protection. **PKCS12KS (PKCS12)** Use this option if your keystore uses the PKCS#12 file format.
alias	The key alias name. The key alias is used by the key locator to find the key within the keystore file.
keyPassword	The key password that is used for recovering the key. This password is needed to access the key object within the keystore file.
keyName	The name of the key. For digital signatures, the key name is used by the request generator or response consumer signing information to determine which key is used to digitally sign the message. For encryption, the key name is used to determine the key used for encryption. The key name must be a fully qualified, distinguished name (DN). For example, `CN=Bob,O=IBM,C=US`.
trustAnchorPath	The file path from which the trust anchor is loaded.

Table 226. Information for the X.509 token (continued). Use the X.509 token to authenticate messages.

Token Information	Description
trustAnchorType	The type of trust anchor.
trustAnchorPassword	The password that is used to check the integrity of the trust anchor or the password used to unlock the keystore.
certStores	A list of certificate stores. A collection certificate store includes a list of untrusted, intermediary certificates and certificate revocation lists (CRLs). The collection certificate store is used to validate the certificate path of the incoming X.509-formatted security tokens.
provider	The security provider.

The following can be specified for a X.509 token:

a. Without any keystore.

b. With a trust anchor. A trust anchor specifies a list of keystore configurations that contain trusted root certificates. These configurations are used to validate the certificate path of incoming X.509-formatted security tokens. For example, when you select the trust anchor or the certificate store of a trusted certificate, you must configure the trust anchor and the certificate store before setting the certificate path.

c. With a keystore that is used for the key locator.

   First, you must have created the keystore file, by using a key tool utility, for example. The keystore is used to retrieve the X.509 certificate. This entry specifies the password that is used to access the keystore file. Keystore objects within trust anchors contain trusted root certificates that are used by the CertPath API to validate the trustworthiness of a certificate chain. The names of the trust anchor and the collection certificate store are created in the certificate path under your token consumer.

d. With a keystore that is used for the key locator and the trust anchor.

e. With a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the consumer token, and represents the token type of this class:

   ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
   Specifies an X.509 certificate token.

   ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509PKIPathv1
   Specifies X.509 certificates in a public key infrastructure (PKI) path. This callback handler is used to create X.509 certificates encoded with the PkiPath format. The certificate is inserted in the Web Services Security header within the SOAP message as a binary security token. A keystore is required for this callback handler. A CRL is not supported by the callback handler; therefore, the collection certificate store is not required or used. If you use this implementation, you must provide a key store password, path, and type on this panel.

   ValueType: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#PKCS7
   Specifies a list of X.509 certificates and certificate revocation lists in a PKCS#7 format. This callback handler is used to create X.509 certificates encoded with the PKCS#7 format. The certificate is inserted in the Web Services Security header in the SOAP message as a binary security token. A keystore is required for this callback handler. You can specify a certificate revocation list (CRL) in the collection certificate store. The CRL is encoded with the X.509 certificate in the PKCS#7 format. If you use this implementation, you must provide a key store password, path, and type.

   For some tokens, WebSphere Application Server provides a predefined local name for the value type. When you specify the following local name, you do not need to specify a value type URI:

   ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
   For an LTPA token, you can use `LTPA` for the value type local name. This local name

causes `http://www.ibm.com/websphere/appserver/tokentype/5.0.2` to be specified for the value type Uniform Resource Identifier (URI).

ValueType: http://www.ibm.com/websphere/appserver/tokentype/5.0.2
For LTPA token propagation, you can use `LTPA_PROPAGATION` for the value type local name. This local name causes`http://www.ibm.com/websphere/appserver/tokentype` to be specified for the value type Uniform Resource Identifier (URI).

8. If the Username token is specified as the token consumer class name, the following token information can be specified:
   a. Whether to specify the nonce.

      This option indicates whether a Nonce is included for the token consumer. Nonce is a unique, cryptographic number that is embedded in a message to help stop repeat, unauthorized attacks of Username tokens. Nonce is valid only when the validating token type is a Username token, and it is available only for the response consumer binding.
   b. Specifies the keyword of the time stamp. This option indicates whether to verify a time stamp in the Username token. The time stamp is valid only when the incorporated token type is a Username token.
   c. Specifies a map that includes key-value pairs. For example, you might specify the value type name and the value type Uniform Resource Identifier (URI). The value type specifies the namespace URI of the value type for the consumer token, and represents the token type of this class:

      URI value type: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
      Specifies a Username token.
9. If a Kerberos v5 token is specified as the token generator class name, the following token information can be specified:

Token Information	Description	Default Value
tokenValueType	Kerberos token value type in QName defined by Oasis Kerberos Token Profile v1.1 specification.	http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ
requireDKT	A boolean value to require a derived key for message protection.	false
clabel	The client label for the derived key.	WS-SecureConversation Specify null to use the default value.
slabel	The service label for the derived key.	WS-SecureConversation Specify null to use the default value.
keylen	The length of the derived key.	16 Specify zero to use the default value
supportTokenRequireSHA1	A boolean value to require a SHA1 key that is used in subsequent request messages when the Kerberos token is used as a supporting token.	false SHA1 key is consumed only if the supporting Kerberos token is protected. If set to true, the SHA1 key is always consumed.

Token Information	Description	Default Value
decComponent	An instance of WSSDecryption .	Set decComponent and verComponent to null to initialize this first for either the decryption or verification component. Then, use the initialized component only in the callback handler constructor for the second component.
verComponent	An instance of WSSVerfication.	Set decComponent and verComponent to null to initialize this first for either the decryption or verification component. Then, use the initialized component only in the callback handler constructor for the second component.

Additional token value types are defined in the OASIS Kerberos Token Profile v1.1 specification. Specify the token value type as the local name. It is not necessary to specify the value type URI for the Kerberos v5 token.

- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ1510
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ4120
- http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ4120

10. If secure conversation is used for message protection, then the following information must be specified:

Information	Description
EncryptionAlgorithm	This determines the key size.
cLabel	The client label used when creating the derived key.
sLabel	The server label used when creating the derived key.

11. Set the components into the wssConsumingContext object.
12. Invoke the wssConsumingContext.process() method.

Results

Using the WSS APIs, you have configured the token consumer.

What to do next

You must specify a similar token generator configuration, if not already completed.

Configuring the consumer security tokens using the WSS API:

You can secure the SOAP messages, without using policy sets, by using the Web Services Security APIs. To configure the token on the consumer side, use the Web Services Security APIs (WSS API). The consumer security tokens are part of the com.ibm.websphere.wssecurity.wssapi.token interface package.

Before you begin

The pluggable token framework in WebSphere Application Server has been redesigned so that the same framework from the WSS API can be reused. The same implementation of creating and validating security token can be used both for the Web Services Security run time and for the WSS API application code. The redesigned framework also simplifies the SPI programming model and will make it easier to add security token types.

You can use the WSS API or you can configure the tokens by using the administrative console. To configure tokens, you must have completed the following token task: configure the generator tokens, as needed.

About this task

On the generator side, the JAAS CallbackHandler and JAAS LoginModule are responsible for creating the security token. The token is created by using the JAAS LoginModule and by using JAAS CallbackHandler to pass authentication data. Then, the JAAS LoginModule creates the securityToken object, such as the UsernameToken, and passes it to the Web Services Security run time.

On the consumer side, the XML format is passed to the JAAS LoginModule for validation or authentication. then the JAAS CallbackHandler is used to pass authentication data from the Web Services Security run time to the LoginModule. After the token is authenticated and a security token object is created, then the token is passed it to the Web Services Security run time.

When using the WSS API for consumer token validation, certain default behaviors occur. The simplest way to use the WSS API is to use the default JAAS login module and callback handler. The example uses the default for them so the example does not specify the JAAS login module name.

The simplest way to use the WSS API is to use the default behavior (see the example code). The WSS API provide defaults for the token type, the token value, and the JAAS configuration name. The default token behaviors include:

Table 227. Default token behaviors. Several token characteristics are configured by default.

Consumer token decisions	Default behavior
Which token type to use	The token type specifies which type of token to use for signing and validating messages. The X.509 token is the default token type. WebSphere Application Server provides the following pre-configured consumer token types: • Security context token • Derived key token • X509 tokens You can also create custom token types, as needed.
What JAAS login configuration name to specify	The JAAS login configuration name specifies which JAAS login configuration name to use.
Which configuration type to use	The JAAS login module configuration type. Only the pre-configured consumer configuration types can be used for consumer token types.

The SecurityToken class (com.ibm.websphere.wssecurity.wssapi.token.SecurityToken) is the generic token class and represents the security token that has methods to get the identity, XML format, and cryptographic keys. Using the SecurityToken class, you can apply both the signature and encryption to the SOAP message. However, to apply both, you must have two SecurityToken objects, one for the signature and one for encryption, respectively.

The following token types are subclasses of the generic security token class:

Table 228. Subclasses of the SecurityToken. Use the subclasses to represent the security token.

Token type	JAAS login configuration name
Security context token	system.wss.consume.sct
Derived key token	system.wss.consume.dkt

The following token types are subclasses of the binary security token class:

Table 229. Subclasses to the BinarySecurityToken. Use the subclasses to represent the binary security token.

Token type	JAAS login configuration name
X.509 token	system.wss.consume.x509
X.509 PKI Path token	system.wss.consume.pkiPath
X.509 PKCS7 token	system.wss.consume.pkcs7

Note:

- For each JAAS login token consumer configuration name, there is a respective token generator configuration name. For example, for the X509Token, the respective token generator configuration name is system.wss.generate.x509.
- The LTPA and LTPA propagation tokens are only available to a requester that is running as a server-based client. The LTPA and LTPA propagation tokens are not supported for the Java SE 6 or Java EE application client.

To validate the X509Token to the SOAP message on the consumer side, the <X509Token> element must be in the <wsse:Security> element.

Procedure

1. To validate the securityToken package, com.ibm.websphere.wssecurity.wssapi.token, first ensure that the application server is installed.
2. *If using the default values*, configures the tokens for the Web Services Security token consumer process. , for each token type, the process is similar to the following token consumer process:
   a. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   b. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   c. Creates a JAAS CallbackHandler with information that is required to validate the security token. Review the token class information for which parameters are required or optional. For example, for an X.509 token, you could configure the following:

Table 230. X.509 token options. Use the X.509 configuration options to control the behavior of the token.

Token Information	Description
keyStoreRef	Indicates the reference name of the keystore that is stored in the cryptographic card. It can be specified when the card is set to the hardware.
keyStorePath	Indicates the path of the keystore file. It is not necessary to specify the keyStorePath if the keyStoreRef is set.
keyStorePassword	Indicates the password of the keystore file.
keyStoreType	Indicates the type of keystore file.
alias	Indicates the alias of the key.
keyPassword	Indicates the password of the key.
keyName	Indicates the subject name of the key.

   d. Sets the callback handler into WSSDecryption, WSSVerification, or WSSConsumingContext.
   e. If the callback handler is set into the WSSDecryption or WSSVerification, adds either one into WSSConsumingContext.
   f. Calls WSSConsumingContext.process().

3. *If using other than the default values*, configures the tokens for the Web Services Security token consumer process. For each token type, the process is similar to the following token consumer process:
   a. If you do not use the default JAAS login module and callback handler, you need to prepare a custom one and register the name of JAAS login configuration using the administrative console in advance.
   b. Uses WSSFactory.getInstance() to get the WSS API implementation instance.
   c. Creates the WSSConsumingContext instance from the WSSFactory instance. Note that the WSSConsumingContext must always be called in a JAX-WS client application.
   d. Creates a callback handler with information that is required to validate the security token. Review the token class information for which parameters are required or optional. For example, for a X.509 token, you can configure the following:

Table 231. X.509 token options. Use the X.509 configuration options to control the behavior of the token.

Token Information	Description
keyStoreRef	Indicates the reference name of the keystore that is stored in the cryptographic card. It can be specified when the card is set to the hardware.
keyStorePath	Indicates the path of the keystore file. It is not necessary to specify the keyStorePath if the keyStoreRef is set.
keyStorePassword	Indicates the password of the keystore file.
keyStoreType	Indicates the type of keystore file.
alias	Indicates the alias of the key.
keyPassword	Indicates the password of the key.
keyName	Indicates the subject name of the key.

   e. Sets JAAS configuration name and callback handler into WSSDecryption or WSSVerification, or WSSConsumingContext.
   f. If JAAS configuration name and callback handler are set into the WSSDecryption or WSSVerification, adds either one into WSSConsumingContext.
   g. Calls WSSConsumingContext.process().

Results

If there is an error condition, a WSSException is provided. If successful, the WSSConsumingContext.process() is called, and the security token on the consumer side is validated (authenticated).

Example

The following sample code provides the WSS API example code for decryption using the default JAAS login module and callback handler:

```
// Get the message context
   Object msgcontext = getMessageContext();

// Generate the WSSFactory instance (step: a)
   WSSFactory factory = WSSFactory.getInstance();

// Generate the WSSConsumingContext instance (step: b)
   WSSConsumingContext gencont = factory.newWSSConsumingContext();

// Generate the callback handler (step: c)
   X509ConsumeCallbackHandler callbackHandler = new
        X509ConsumeCallbackHandler(
                                   "",
                                   "enc-sender.jceks",
                                   "jceks",
                                   "storepass".toCharArray(),
                                   "alice",
                                   "keypass".toCharArray(),
                                   "CN=Alice, O=IBM, C=US");

// Generate the WSSDecryption instance (step: d)
   WSSDecryption dec = factory.newWSSDecryption(X509Token.class,
```

```
                                        callbackHandler);

// Add WSSDecryption to WSSConsumingContext (step: e)
   concont.add(dec);

// Validate the WS-Security header (step: f)
concont.process(msgcontext);
```

What to do next

For each token type, configure the token using the WSS APIs or using the administrative console. Next, specify the similar generator tokens if you have not done so.

If both the generator and consumer tokens are configured, continue securing SOAP messages at the response consumer using the WSS APIs or configure the tokens using the administrative console.

If both the generator and consumer tokens are configured, continue securing SOAP messages either by verifying the signature or by decrypting the message, as needed. You can use either the WSS APIs or the administrative console to secure the SOAP messages.

Configuring Web Services Security using the WSS APIs:

The Web Services Security application programming interfaces (WSS API) provide support for securing SOAP message.

Before you begin

Web Service Security supports the following programming models:

- Programming API for securing SOAP message with Web Services Security (WSS API).

  The API programming model design has been redesigned. The new design is an interface-based programming model and is based on Web Services Security Version 1.1 standards but the design also includes support for Web Services Security Version 1.0 for securing the SOAP message. The WSS API programming model implementation is a simplified version, which is based on an early draft proposal of JSR-183, which is the JSR for defining Java API binding for Web Services Security. By design, because the application code is programmed to the interface, any application code that is programmed with the open source implementation should be able to run on the WebSphere Application Server with minimal changes or no changes at all.
- Service Programming Interfaces (SPI) for a service provider

  Similarly, the Web Services Security run time token generation and token consuming SPI have been redesign so that the same security token interface and JAAS Login Module implementation can be used for both the WSS API and the SPI. The WSS SPI for the service provider extend the security token types and provide keys and deriving keys for signing, signature verification, encryption and decryption.

Usage statement: You must use the IBM implementation of the WS-Security standards in the context of web services.

About this task

These programming models extend the following functions :

- Security token types and deriving keys for signing
- Signature and verification
- Encryption and decryption

The following figure demonstrates how to use the simplified WSS APIs to secure a SOAP message by using XML digital signature and XML encryption.

The configuration model for web services has also been redesigned from a deployment descriptor model to a policy set model. The configuration programming model is based on configuring policy sets using a security policy to specify security constraints.

The functions provided by the policy set configurations are the same as the functions supported by the WSS API for the Web Services Security run time. However, the security policy that is defined using policy sets has a higher priority over the WSS API. When the WSS API and the policy set are both used in the application, the default behavior is for the security policy from the policy set to be enforced and the WSS API to be ignored. To use the WSS API in the application, you must make sure that there is no policy set attached to the application or to the application resources, or make sure there is no security policy in the attached policy set.

Web Service Security can be enabled by either using a policy set that is configured by using the administrative console, or by using the WSS API for configuration.

Using the WSS API, complete the following high-level steps to secure the SOAP message:

Procedure

1. Use the WSSSignature API to configure the signing information for the request generator (client side) binding. Different message parts can be specified in the message protection for a request on the generator side. The default required parts are BODY, ADDRESSING_HEADERS, and TIMESTAMP.

   The WSSSignature API also specifies the different algorithm methods to be used with the signature for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.
2. Use the WSSSignPart API if you want to add or change the signed parts to be used for message protection. The default signed parts are WSSSignature.BODY, WSSSignature.ADDRESSING_HEADERS, and WSSSignature.TIMESTAMP.

   The WSSSignPart API also specifies the different algorithm methods to be used if you added or changed the signed parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N. For example, use the WSSSignPart API if you want to generate the signature for the SOAP message using the SHA256 digest method instead of the default value of SHA1.
3. Use the WSSEncryption API to configure the encryption information on the request generator side. The encryption information on the generator side is used for encrypting an outgoing SOAP message for the request generator (client side) bindings. The default targets of encryption are BODY_CONTENT and SIGNATURE

   The WSSEncryption API also specifies the different algorithm methods to be used to protect message confidentiality. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.
4. Use the WSSEncryptPart API if you want to add or change the encrypted parts to be used for message confidentiality. For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

   No algorithm methods are required for encrypted parts.
5. Use the WSS API to attach the token on the generator side. The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for creating the security token on the generator side. Different stand-alone tokens can be sent in request or response. The default token is the X509Token. The other token that can be used for signing is the DerivedKeyToken, which is used only with Web Services Secure Conversation (WS-SecureConversation).
6. Use the WSSVerification API to verify the signature for the response consumer (client side) binding. Different message parts can be specified in the message protection for a response on the consumer side. The required targets for verification are BODY, ADDRESSING_HEADERS, and TIMESTAMP.

The WSSVerification API also specifies the different algorithm methods to be used for verifying the signature and for message protection. The default signature method is RSA_SHA1. The default canonicalization method is EXC_C14N.

7. Use the WSSVerifyPart API to add or change the verify signed parts to be used for message protection. The required verify parts are WSSVerification.BODY, WSSVerification.ADDRESSING_HEADERS, and WSSVerification.TIMESTAMP.

   The WSSVerifyPart API also specifies the different algorithm methods to be used if you added or changed the verification parts. The default digest method is SHA1. The default transform method is TRANSFORM_EXC_C14N.

8. Use the WSSDecryption API to configure the decryption information for the response consumer (client side) binding. The decryption information on the consumer side is used for decrypting an incoming SOAP message. The default targets of decryption are BODY_CONTENT and SIGNATURE. The default data encryption method is AES128. The default key encryption method is KW_RSA_OAEP.

   No algorithm methods are required for decryption.

9. Use the WSSDecryptPart API if you want to add or change the decrypted parts to be used for message confidentiality. For example, if you want to change the data encryption method from the default value of AES128 to TRIPLE_DES.

   No algorithm methods are required for decrypted parts.

10. Use the WSS API to configure the token on the consumer side. The requirements for the security token depend on the token type. The JAAS Login Module and the JAAS CallbackHandler are responsible for validating (authenticating) the security token on the consumer side. Different stand-alone tokens can be sent in request or response.

    The WSS API adds the information for the candidate token that is used for decryption. The default token is X509Token.

Results

What to do next

The Web Services Security run time token generation and token consuming Service Programming Interfaces (SPI) have been redesign so that the same Security Token interface and JAAS Login Module implementation can be used in both the WSS API and the SPI. See the SPI information for detail descriptions.

Web Services Security APIs:

The Web Services Security programming model provides application programming interfaces (WSS API) for securing the SOAP message. The WSS API model is based on Web Services Security Version 1.1 standards but also includes support for Web Services Security Version 1.0.

The Web Services Security APIs (WSS APIs) can generate and process the following SOAP-related bindings for XML security:

- XML signature and signature verification
- XML encryption and decryption

The token processing and pluggable token architecture in the Web Service Security run time has been redesign to reuse the same Security Token interface and the JAAS Login Module as those used for the WSS APIs.

The following table lists the WSS API interfaces that are provided with WebSphere Application Server and used to configure signing and encryption information in the SOAP bindings for the generator and consumer bindings.

Table 232. WSS API interfaces. Use the interfaces to configure security information in the bindings.

WSS API interfaces	Description
WSSDecryption	Package: com.ibm.websphere.wssecurity.wssapi.decryption This interface is responsible for specifying decryption. The default values for decryption include: • Targets: BODY_CONTENT, SIGNATURE • Data encryption method: AES128 • Key encryption method: KW_RSA_OAEP • Security token: X.509
WSSDecryptPart	Package: com.ibm.websphere.wssecurity.wssapi.decryption This interface is responsible for adding decrypted parts, as needed. If specified, the default values for decrypted parts include: • Security token: X.509 • Transform method: N/A (not applicable)
WSSEncryption	Package: com.ibm.websphere.wssecurity.wssapi.encryption This interface is responsible for the encryption component. The default values for encryption include: • Targets: BODY_CONTENT, SIGNATURE • Data encryption method: AES128 • Key encryption method: KW_RSA_OAEP • Security token: X.509 • refType: SecurityToken.REF_KEYID • mtomOptimize: false
WSSEncryptPart	Package: com.ibm.websphere.wssecurity.wssapi.encryption This interface is responsible for adding encrypted parts, as needed. If specified, the default values for encrypted parts include: • Transform method: N/A (not applicable)
WSSSignature	Package: com.ibm.websphere.wssecurity.wssapi.signature This interface is responsible for specifying the signature. The default values for signature include: • Targets: BODY, ADDRESSING_HEADERS, TIMESTAMP • Signature method: RSA_SHA1 • Canonicalization method: EXC_C14N • Security token: X.509 • Type of token reference: SecurityToken.REF_STR
WSSSignPart	Package: com.ibm.websphere.wssecurity.wssapi.signature This interface is responsible for adding signed parts, as needed. If specified, the default values for signed parts include: • Transform method : TRANSFORM_EXC_C14N • Digest method: SHA1
WSSVerification	Package: com.ibm.websphere.wssecurity.wssapi.verification This interface is responsible for specifying the signature verification. The default values for verification include: • Targets: BODY, ADDRESSING_HEADERS, TIMESTAMP • Signature method: RSA_SHA1 • Canonicalization method: EXC_C14N • Security token: X.509
WSSVerifyPart	Package: com.ibm.websphere.wssecurity.wssapi.verification This interface is responsible for adding verify parts, as needed. If specified, the default values for verify parts include: • Digest method: SHA1 • Transform method: TRANSFORM_EXC_C14N

Also see the information about pre-configured generator and consumer tokens.

Web Services Security configuration considerations when using the WSS API:

To secure Web Services Security for WebSphere Application Server, you can specify several different configurations using the Web Services Security APIs (WSS API). The Web Services Security specification provides a flexible way to secure web services messages using XML digital signature, XML encryption, and attaching security tokens. You can enable Web Services Security by either configuring a policy set or by using the Web Services Security APIs (WSS API). The implementation for WSS API has default values for which message parts are to be signed or encrypted. The default values for the WSS APIs help end users to enable Web Services Security quickly.

Different message parts can be specified in the message protection for request or response, and different stand-alone tokens can be sent in request or response. However, there is only one symmetric or one asymmetric binding assertion to describe the token type and the algorithm that is used for message protection.

Using the WSS API, you can override any default values. However, when you alter the protection parts, note that all the default protection parts are cleared. For example, if you specify that you want to encrypt the Username token instead of the default X.509 token, all the default values of the encrypting protection parts are cleared.

The following table shows an example of the relationships between each of the configurations:

Table 233. Request generator and response consumer configurations. Use the table to determine the mapping between the configurations and the default values.

Type of configuration	Configuration name	Configurations and default values
Request generator	Signing information	• Canonicalization method: WSSSignature.EXC_C14N • Signature method: WSSSignature.RSA_SHA1 • Digest method: WSSSignPart.SHA1 • Transform method: WSSSignPart.TRANSFORM_EXC_C14N • Signed part - Body: WSSSignature.BODY • Signed part - Addressing: WSSSignature.ADDRESSING_HEADERS • Signed part - Timestamp: WSSSignature.TIMESTAMP • Token reference: SecurityToken.REF_STR • Token - Value type: X509Token.ValueType • Token - JAAS login configuration name: system.wss.generate.x509
Response consumer	Signature verification information	• Canonicalization method: WSSVerification.EXC_C14N • Signature method: WSSVerification.RSA_SHA1 • Transform method: WSSVerifyPart.TRANSFORM_EXC_C14N • Signed part - Body: WSSVerification.BODY • Signed part - Addressing: WSSVerification.ADDRESSING_HEADERS • Signed part - Timestamp: WSSVerification.TIMESTAMP • Token - Value type: X509Token.ValueType • Token - JAAS login configuration name: system.wss.consume.x509
Request generator	Encryption information	• Encrypted key: true • Key encryption method: WSSEncryption.KW_RSA_OAEP • Data encryption method: WSSEncryption.AES128 • Encryption part: WSSEncryption.BODY_CONTENT • Token reference: SecurityToken.REF_KEYID • Token - Value type: X509Token.ValueType • Token - JAAS login configuration name: system.wss.generate.x509
Response consumer	Decryption information	• Encrypted key: true • Key decryption method: WSSDecryption.KW_RSA_OAEP • Data decryption method: WSSDecryption.AES128 • Decryption part: WSSDecryption.BODY_CONTENT • Token - Value type: 509Token.ValueType • Token - JAAS login configuration name: system.wss.consume.x509

Encrypted SOAP headers:

The encrypted header element provides a standard way of encrypting SOAP headers. As one of the extensions to the OASIS SOAP message security specification, the encrypted header element indicates that the responder has processed the request. Encrypting SOAP headers and parts help to provide more secure message-level security.

The EncryptedHeader or <wsse11:EncryptedHeader> element is a part of the updated Web Services Security Version 1.1 standard and enables interoperability with other vendors that support the Version 1.1 standards, such as Microsoft .NET and DataPower®.

Use the EncryptedHeader element for encrypting SOAP header blocks. The EncryptedHeader element allows Web Services Security to be compliant with the SOAP mustUnderstand processing guidelines and to prevent disclosure of information that is contained in attributes on a SOAP header block.

The <wsse11:EncryptedHeader> element must contain one <xenc:EncryptedData> element. Only one <xenc:EncryptedData> element per encrypted header element is permitted.

Encrypted data element

Normally, the programming model, such as JAX-WS, deserializes the SOAP message to a Java binding object before dispatching the call to the application code. However, if the SOAP message is encrypted, the deserialization fails because, before encryption, the original content is replaced with the EncryptedData XML element from the XML Encryption standard.

In certain cases, it might be desirable for the token that is included in the <wsse:Security> header to be encrypted for the recipient processing role.

Follow these guidelines when using the EncryptedData element:

- The EncryptedHeader element must contain one EncryptedData element.
- The <xenc:EncryptedData> element may be used to contain a security token and include it in the <wsse:Security> header.
- The <xenc:EncryptedData> must not include an XML ID for referencing the contained security token.
- All <xenc:EncryptedData> tokens must either have an embedded encryption key or must be referenced by a separate encryption key.
- If compliance with Basic Security Profile 1.1 is desired, the <xenc:EncryptedData> element must have an Id attribute.

Policy assertion for encrypted parts

The EncryptedParts policy assertion specifies which header is to be encrypted in the security policy. The following table describes the elements and attributes that can be used for EncryptedParts.

Table 234. Attributes and elements of the EncryptedParts element. Use encrypted parts to provide more secure message-level security.

Element or attribute	Description
/sp:EncryptedParts/sp:Header	Optional. Presence of this optional element indicates that a specific SOAP header (or set of such headers) must be protected. You can have multiple sp:Header elements within a single EncryptedParts element. Each header (or set of headers) must be encrypted, and this encryption will encrypt the elements by using Web Services Security Version 1.1 encrypted headers. As such, if WS-Secuirty 1.1 Encrypted Headers are not supported by a service, then the headers cannot be encrypted by using message-level security. If multiple SOAP headers with the same local name but different namespace names are to be encrypted, multiple sp:Header elements are required, either as part of a single sp:EncryptedParts assertion or as part of separate sp:EncryptedParts assertions.

Table 234. Attributes and elements of the EncryptedParts element (continued). Use encrypted parts to provide more secure message-level security.

Element or attribute	Description
/sp:EncryptedParts/sp:Header/@Name	Optional. This attribute indicates the local name of the SOAP header to be confidentiality protected. If this attribute is not specified, all SOAP headers whose namespace matches the Namespace attribute are to be protected.
/sp:EncryptedParts/sp:Header/@Namespace	Required. This attribute indicates the namespace of the SOAP headers to be confidentiality protected.

The following message example shows what the EncryptedHeader element looks like on a message where the EncryptedParts policy assertion for the encrypted header has been specified on the policy:

```
<S:Envelope xmlns:S="..." xmlns:wsse="..." xmlns:wsse11="..." xmlns:wsu="..."
   xmlns:xenc="..." xmlns:ds="...">
   <S:Header>
     <wsse:Security>
      <!-- Tokens etc. -->
       <xenc:EncryptedKey>
         <xenc:EncryptionMethod Algorithm="..."/>
           <ds:KeyInfo>
             ... ...
           </ds:KeyInfo>
          <xenc:CipherData>
            <xenc:CipherValue>...</xenc:CipherValue>
          </xenc:CipherData>
          <xenc:ReferenceList>
            <xenc:DataReference URI="#hdrID"/>
          </xenc:ReferenceList>
        </xenc:EncryptedKey>
     </wsse:Security>
     <wsse11:EncryptedHeader wsu:Id="hdrID">
       <xenc:EncryptedData Id="encDataID">
         <xenc:CipherData>
           <xenc:CipherValue>...</xenc:CipherValue>
         </xenc:CipherData>
         ...
       </xenc:EncryptedData>
     </wsse11:EncryptedHeader>
   </S:Header>
   <S:Body>
     ...
   </S:Body>
</S:Envelope>
```

To encrypt headers in the Web Services Security Version 1.0 specification format, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.0 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the target header for encryption is replaced by an <EncryptedData> element, instead of an <EncryptedHeader> element that contains an <EncryptedData> element.

For Web Services Security Version 1.1 behavior that is equivalent to WebSphere Application Server versions prior to version 7.0, specify the com.ibm.wsspi.wssecurity.encryptedHeader.generate.WSS1.1.pre.V7 property with a value of `true` on the <encryptionInfo> element in the binding. When this property is specified, the <EncryptedHeader> element includes a wsu:Id parameter and the <EncryptedData> element omits the Id parameter. This property should only be used if compliance with Basic Security Profile 1.1 is not required.

For complete information about the EncryptedHeader element and the EncryptedData element, see the Web Services Security Version 1.1 specification.

Signature confirmation:

Web Services Security signature confirmation is an enhanced XML digital signature, and it is included in the Web Services Security standard. XML digital signature is used for signing elements of the SOAP envelope.

As one of the extensions to the OASIS SOAP message security specification, the signature confirmation element incorporates the elements that are needed within the response message in order to confirm the signature that is contained in a request message. XML digital signature and signature confirmation help to provide more secure message-level security.

Web Services Security Version 1.0 for SOAP message security did not provide any guidance on how to confirm mutual understanding of the request that prompted this response. The SignatureConfirmation or <wsse11:SignatureConfirmation> element has been added to the Web Services Security Version 1.1 specification. The <wsse11:SignatureConfirmation> element ensures that the signature is processed by the intended recipient and indicates that the responder has processed the signature in the request. The signature confirmation element is part of the updated Web Services Security standard and enables interoperability with other vendors that support the Version 1.1 standards, such as Microsoft .NET and DataPower.

Because of the stateless nature of web services and due to different message exchange patterns (MEPs), consider the following assumptions:

- Assume that session affinity is enabled if a cluster is enabled for the clients that are running in WebSphere Application Server. When session affinity is enabled, it implies that the response is sent back to the initiating client of the server.
- Assume WS-Addressing is enabled for asynchronous message exchange patterns. When WS-Addressing is enabled, it allows the run time to relate the response back to the request. An asynchronous response is sent back to the application of the initiating WebSphere Application Server.

Syntax

The SignatureConfirmation element indicates that the responder has processed the signature in the request. When this element is not present in a response, the initiator interprets that the responder is not compliant.

The format for the signature confirmation element is as follows:

```
<wsse11:SignatureConfirmation wsu:Id="..." Value="..." />
```

where:

wsu:Id
The identifier that is used when referencing this element in the <ds:SignedInfo> reference list of the signature of the associated response message. This attribute is required so that unambiguous references are made to this <wsse11:SignatureConfirmation> element.

Value This attribute is optional and contains the contents of a <ds:SignatureValue> that is copied from the associated request. If the request is unsigned, this attribute must not be present. If this attribute is specified without a value (empty), the initiator interprets this as incorrect behavior and processes it accordingly. When this attribute is not present, the initiator interprets this to mean that the response is based on a request that was not signed.

Configuration

To configure signature confirmation, configure the policy file using the administrative console, and select **Require signature confirmation**. To process Signature Confirmation correctly, the initiator of the request needs to preserve the signatures during request generator processing and later needs to retrieve the signatures for confirmation checks.

Response generation rules

Additional SOAP security elements for the SOAP responder are used to confirm that the response is in relationship to a particular request. The responder must include the contents of the <ds:SignatureValue> element of the request signature as the value of the @Value attribute of the <wsse11:SignatureConfirmation> element.

The following response generation rules apply when using the SignatureConfirmation policy assertion:

- If there are no signatures on the request, the response contains one SignatureConfirmation element, without a value. For MEPs where there are multiple requests (all without signatures) and one response, the response contains one SignatureConfirmation element without a value.
- If there are signatures on the request, the response contains a SignatureConfirmation element for each signature, with a value that matches the signature value on the request. For MEPs where there are multiple requests, with at least one containing a signature, and one response, the response contains a SignatureConfirmation element for each signature that is found on the requests, with a value that matches the signature value on the request.
- For MEPs where there is one request and multiple responses, each response contains the appropriate SignatureConfirmation elements as noted in the first and second bullets.
- If the SOAP request contains multiple signatures, the requester will find all of the signature confirmation elements contained in the response, and will check the values of the value fields of the signature confirmation elements against the values of the signatures in the original SOAP request.

Developing JAX-WS based web services client applications that retrieve security tokens

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers.

About this task

This information applies only to Java API for XML-based Web Services (JAX-WS) .

The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

With a client application, the application serves as the request generator and the response consumer and runs as the Java Platform, Enterprise Edition (Java EE) client application. The consumer component for Web Services Security stores the security tokens that it receives in one of the properties of the MessageContext object for the current web services call. You can retrieve a set of token objects through the javax.xml.rpc.Stub interface of that web services call. You must know which security tokens to retrieve and their token IDs in case multiple security tokens are included in the SOAP security header. Complete the following steps to retrieve the security token data from a client application:

Procedure

1. Use the com.ibm.wsspi.wssecurity.token.tokenPropergation key string to obtain the Hashtable for the tokens through a property value in the javax.xml.ws.Stub interface. The following example shows how to obtain the Hashtable:

```
java.util.Hashtable t;

javax.xml.ws.Service serv = ...;
serv.addPort(...);
javax.xml.ws.Dispatch<Object> dispatch = svc.createDispatch(...);

Map<String, Object> requestContext = dispatch.getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, ..);
requestContext.put(BindingProvider.SOAPACTION_USE_PROPERTY, ..);
```

```
requestContext.put(BindingProvider.SOAPACTION_URI_PROPERTY, ..);

String response = dispatch.invoke(body.toString());

Map<String, Object> responseContext = dispatch.getResponseContext();

t = (Hashtable) responseContext.get(
com.ibm.wsspi.wssecurity.Constants.WSSECURITY_TOKEN_PROPERGATION);
```

2. Search the targeting token objects in the Hashtable. Each token object in the Hashtable is set with its token ID as a key. You must have prior knowledge of the security token IDs to retrieve the security tokens. The following example shows how to retrieve a username token from the security header with a certain token ID value:

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
if (t != null) {
  unt = (com.ibm.wsspi.wssecuty.auth.token.UsernameToken)t.get("...");
}
```

Results

After completing these steps, you have retrieved the security tokens that are processed by the Web Services Security handler in a client application.

Developing JAX-WS based web services server applications that retrieve security tokens

With a server application, the application acts as the request consumer, and the response generator is deployed and runs in the Java Platform, Enterprise Edition (Java EE) container. The consumer component for Web Services Security stores the security tokens that it receives in the Java Authentication and Authorization Service (JAAS) Subject of the current thread. You can retrieve the security tokens from the JAAS Subject that is maintained as a local thread in the container.

About this task

This information applies only to Java API for XML-based Web Services (JAX-WS).

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers. The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

Complete the following steps to retrieve the security token data from a server application:

Procedure

1. Obtain the JAAS Subject of the current thread using the WSSubject API. If you enable Java 2 Security on the Global security panel in the administrative console, access to the JAAS Subject is denied if the application code is not granted the javax.security.auth.AuthPermission("wssecurity.getCallerSubject") permission. The following code sample shows how to obtain the JAAS subject:

```
javax.security.auth.Subject subject;

try {
subject = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
} catch (com.ibm.websphere.security.WSSecurityException e) {
  ...
}
```

2. Obtain a set of private credentials from the Subject. For more information, see the application programming interface (API) com.ibm.websphere.security.auth.WSSubject class through the information center . To access this information within the information center, click **Reference >**

Developer > API Documentation > Application Programming Interfaces. In the Application Programming Interfaces article, click **com.ibm.websphere.security.auth > WSSubject**.

Attention: When Java 2 Security is enabled, you might need to use the AccessController class to avoid a security violation that is caused by operating the security objects in the Java EE container.

The following code sample shows how to set the AccessController class and obtain the private credentials:

```
Set s = (Set) AccessController.doPrivileged(new PrivilegedAction() {
   public Object run() {
   return subj.getPrivateCredentials();
   }
      });
```

3. Search the targeting token class in the private credentials. You can search the targeting token class by using the java.util.Iterator interface. The following example shows how to retrieve a username token with a certain token ID value in the security header. You can also use other method calls to retrieve security tokens. For more information, see the application programming interface (API) documents for the com.ibm.wsspi.wssecurity.auth.token.Token interface or custom token classes.

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
Iterator it = s.iterator();
while (it.hasNext()) {
  Object obj = it.next();
  if (obj != null &&
obj instanceOf com.ibm.wsspi.wssecurity.auth.token.UsernameToken) {
    unt =(com.ibm.wsspi.wssecurity.auth.token.UsernameToken) obj;
if (unt.getId().equals("...")) break;
else continue;
  }
}
```

Results

After completing these steps, you have retrieved the security tokens from the JAAS Subject in a server application.

Developing message-level security for JAX-RPC web services

IBM® WebSphere® Application Server supports the Java™ API for XML-Based Web Services (JAX-WS) programming model and the Java API for XML-based RPC (JAX-RPC) programming model.

Developing web services clients that retrieve tokens from the JAAS Subject in an application

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers.

About this task

This information applies only to Java API for XML-based RPC (JAX-RPC) Web services.

The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

With a client application, the application serves as the request generator and the response consumer and runs as the Java Platform, Enterprise Edition (Java EE) client application. The consumer component for Web Services Security stores the security tokens that it receives in one of the properties of the MessageContext object for the current Web services call. You can retrieve a set of token objects through the javax.xml.rpc.Stub interface of that web services call. You must know which security tokens to retrieve and their token IDs in case multiple security tokens are included in the SOAP security header. Complete the following steps to retrieve the security token data from a client application:

Procedure

1. Use the com.ibm.wsspi.wssecurity.token.tokenPropergation key string to obtain the Hashtable for the tokens through a property value in the javax.xml.rpc.Stub interface. The following example shows how to obtain the Hashtable:

```
java.util.Hashtable t;
javax.xml.rpc.Service serv = ...;
MyWSPortType pt = (MyWSPortType)serv.getPort(MyWSPortType.class);
t = (Hashtable)((javax.xml.rpc.Stub)pt)._getProperty(
com.ibm.wsspi.wssecurity.Constants.WSSECURITY_TOKEN_PROPERGATION);
```

2. Search the targeting token objects in the Hashtable. Each token object in the Hashtable is set with its token ID as a key. You must have prior knowledge of the security token IDs to retrieve the security tokens. The following example shows how to retrieve a username token from the security header with a certain token ID value:

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
if (t != null) {
  unt = (com.ibm.wsspi.wssecuty.auth.token.UsernameToken)t.get("...");
}
```

Results

After completing these steps, you have retrieved the security tokens from the JAAS Subject in a client application

Developing web services applications that retrieve tokens from the JAAS Subject in a server application

With a server application, the application acts as the request consumer, and the response generator is deployed and runs in the Java Platform, Enterprise Edition (Java EE) container. The consumer component for Web Services Security stores the security tokens that it receives in the Java Authentication and Authorization Service (JAAS) Subject of the current thread. You can retrieve the security tokens from the JAAS Subject that is maintained as a local thread in the container.

About this task

This information applies only to Java API for XML-based RPC (JAX-RPC) Web services.

The security handlers are responsible for propagating security tokens. These security tokens are embedded in the SOAP security header and passed to downstream servers. The security tokens are encapsulated in the implementation classes for the com.ibm.wsspi.wssecurity.auth.token.Token interface. You can retrieve the security token data from either a server application or a client application.

Complete the following steps to retrieve the security token data from a server application:

Procedure

1. Obtain the JAAS Subject of the current thread using the WSSubject utility class. If you enable Java 2 Security on the Global security panel in the administrative console, access to the JAAS Subject is denied if the application code is not granted the javax.security.auth.AuthPermission("wssecurity.getCallerAsSubject") permission. The following code sample shows how to obtain the JAAS subject:

```
javax.security.auth.Subject subj;
try {
subj = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
} catch (com.ibm.websphere.security.WSSecurityException e) {
  ...
}
```

2. Obtain a set of private credentials from the Subject. For more information, see the application programming interface (API) com.ibm.websphere.security.auth.WSSubject class through the information center . To access this information within the information center, click **Reference >**

Developer > API Documentation > Application Programming Interfaces. In the Application Programming Interfaces article, click **com.ibm.websphere.security.auth > WSSubject**.

Attention: When Java 2 Security is enabled, you might need to use the AccessController class to avoid a security violation that is caused by operating the security objects in the Java EE container.

The following code sample shows how to set the AccessController class and obtain the private credentials:

```
Set s = (Set) AccessController.doPrivileged(new PrivilegedAction() {
   public Object run() {
   return subj.getPrivateCredentials();
   }
      });
```

3. Search the targeting token class in the private credentials. You can search the targeting token class by using the java.util.Iterator interface. The following example shows how to retrieve a username token with a certain token ID value in the security header. You can also use other method calls to retrieve security tokens. For more information, see the application programming interface (API) documents for the com.ibm.wsspi.wssecurity.auth.token.Token interface or custom token classes.

```
com.ibm.wsspi.wssecurity.auth.token.UsernameToken unt;
Iterator it = s.iterator();
while (it.hasNext()) {
  Object obj = it.next();
  if (obj != null &&
obj instanceOf com.ibm.wsspi.wssecurity.auth.token.UsernameToken) {
    unt =(com.ibm.wsspi.wssecurity.auth.token.UsernameToken) obj;
if (unt.getId().equals("...")) break;
else continue;
  }
}
```

Results

After completing these steps, you have retrieved the security tokens from the JAAS Subject in a server application

Web Services Security service provider programming interfaces

Several Service Provider Interfaces (SPIs) are provided to extend the capability of the Web Services Security runtime.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

The following list contains the SPIs that are available for WebSphere Application Server:

Procedure

- com.ibm.wsspi.wssecurity.config.KeyLocator is an abstract for obtaining the keys for digital signature and encryption. The following list contains the default implementations:
  1. com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator implements the Java key store.
  2. com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator povides a mapping of the authenticated identity to a key for encryption or, the implementation uses the default key that is specified.
  3. com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator Provides the capability of using the signer key for encryption in the response message. This implementation is typically used in the response sender configuration.

- com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator is an interface that is used to evaluate the trust for identity assertion. The default implementation is com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl, which enables you to define a list of trusted identities.
- The Java Authentication and Authorization Service (JAAS) CallbackHandler application programming interfaces (APIs) are used for token generation by the request sender. This interface can be extended to generate a custom token that can be inserted in the Web Services Security header. The following list contains the default implementations that are provided by WebSphere Application Server:
  1. com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler presents a login prompt to gather the basic authentication data. Use this implementation in the client environment only.
  2. com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler collects the basic authentication data in the standard in (stdin) prompt. Use this implementation in the client environment only.

     Restriction: If you have a multi-threaded client and multiple threads attempt to read from standard in at the same time, all the threads will not successfully obtain the user name and password information. Therefore, you cannot use the com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler implementation with a multi-threaded client where multiple threads might attempt to obtain data from standard in concurrently.
  3. com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler reads the basic authentication data from the application binding file. This implementation might be used on the server side to generate a user name token.
  4. com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler Generates a Lightweight Third Party Authentication (LTPA) token in the Web Services Security header as a binary security token. If basic authentication data is defined in the application binding file, it is used to perform a login, to extract the LTPA token from the WebSphere credentials, and to insert the token in the Web Services Security header. Otherwise, it will extract the LTPA security token from the invocation credentials (RunAs identity) and insert the token in the Web Services Security header.

What to do next

The JAAS LoginModule API is used for token validation on the request receiver side of the message. You can implement a custom LoginModule API to perform validation of the custom token on the request receiver of the message. After the token is verified and validated, the token is set as the caller and then run as the identity in the WebSphere Application Server runtime. The identity is used for authorization checks by the containers before a Java Platform, Enterprise Edition (Java EE) resource is invoked. The following list presents the default AuthMethod configurations provided by WebSphere Application Server:

BasicAuth
Validates a user name token.

Signature
Maps the distinguished name (DN) of a verified certificate to a Java Authentication and Authorization Service (JAAS) subject.

IDAssertion
Maps a trusted identity to a JAAS subject.

LTPA Validates an LTPA token that is received in the message and creates a JAAS subject.

Configuring Web Services Security during application assembly

If you configure Web Services Security with an assembly tool, the Web Services Security binding information is modified

Configuring HTTP outbound transport level security with an assembly tool

You can configure the HTTP outbound transport level security with an assembly tool.

Before you begin

You can configure HTTP outbound transport level security with assembly tools provided with WebSphere Application Server.

This task is one of several ways that you can configure the HTTP outbound transport level security for a web service acting as a client to another web service server. You can also configure the HTTP outbound transport level security with the administrative console or by using the Java properties. If you do not configure the HTTP outbound transport level security, the web services runtime defers to the Java Platform, Enterprise Edition (Java EE) security runtime in the WebSphere product for an effective Secure Sockets Layer (SSL) configuration. If there is no SSL configuration with the Java EE security runtime in the WebSphere product, the Java Secure Socket Extension (JSSE) system properties are used.

About this task

If you configure the HTTP outbound transport level security with assembly tool or with the administrative console, the Web Services Security binding information is modified. If you have not yet installed the web services application into WebSphere Application Server, you can configure the HTTP SSL configuration with an assembly tool. This task assumes that you have not deployed the web services application into the WebSphere product.

If you configure the HTTP outbound transport level security using the standard Java properties for JSSE, the properties are configured as system properties. The configuration that is specified in the binding takes precedence over the Java properties. However, the configurations that are specified by the Java EE security programming model, or are associated with the Dynamic selection, have a higher precedence.

To learn more, see the secure communications using Secure Sockets Layer information.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the HTTP outbound transport level security. Read about enabling web service endpoints in the Rational Application Developer documentation.

Results

You have configured the HTTP outbound transport level security for a web service acting as a client to another web service with an assembly tool.

Configuring HTTP basic authentication for JAX-RPC web services with an assembly tool

You can configure HTTP basic authentication for Java API for XML-based RPC (JAX-RPC) web services with an assembly tool.

Before you begin

You can configure HTTP basic authentication with assembly tools provided with WebSphere Application Server.

About this task

This task is one of three ways that you can configure HTTP basic authentication. You can also configure HTTP basic authentication with the administrative console or by modifying the HTTP properties programmatically.

If you choose to configure the HTTP basic authentication with an assembly tool or with the administrative console, the Web Services Security binding information is modified. You can use an assembly tool to configure HTTP basic authentication before you deploy or install the web services application into WebSphere Application Server. This task assumes that you have not deployed the web services application into the WebSphere product.

If you configure HTTP basic authentication programmatically, the properties are configured in the Stub or Call instance. The values set programmatically take precedence over the values defined in the binding.

The HTTP basic authentication that is discussed in this topic is orthogonal to WS-Security and is distinct from basic authentication that WS-Security supports. WS-Security supports basic authentication token, not HTTP basic authentication.

To configure HTTP basic authentication, use the WebSphere Application Server tools to modify the binding information.

Procedure

1. Start an assembly tool. Read about starting the assembly tool in the Rational Application Developer documentation.
2. If you have not done so already, configure the assembly tool so that it works on Java EE modules. You need to make sure that the **Java EE** and **Web** categories are enabled. Read about configuring the assembly tool in the Rational Application Developer documentation.
3. Migrate the web application archive (WAR) files that are created with the Assembly Toolkit, Application Assembly Tool (AAT) or a different tool to the Rational Application Developer assembly tool. To migrate files, import your WAR files to the assembly tool. Read about migrating code artifacts to an assembly tool in the Rational Application Developer documentation.
4. Configure the HTTP basic authentication in the Web Services Client Port Binding page for a web service or a web service client. The Web Services Client Port Binding page is available after double-clicking the client deployment descriptor file. Read about Web Services Client Port Bindings in the Rational Application Developer documentation.

Configuring XML digital signature for Version 5.x web services with an assembly tool

XML digital signature is one of the methods WebSphere® Application Server provides to secure your web services. It provides message integrity and authentication capabilities when used with SOAP messages.

Configuring trust anchors using an assembly tool

Use an assembly tool to configure trust anchors (that specify keystores which contain trusted root certificates to validate the signer certificate) or trust stores at the application level.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. You can configure an application-level trust anchor using an assembly tool or the administrative console. This document describes how to configure the application-level trust anchor using an assembly tool.

About this task

A trust anchor specifies keystores that contain trusted root certificates, which validate the signer certificate. These keystores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `application-client.xml` file when web services are acting as client) to validate the signer certificate of the digital signature. The keystores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `application-client.xml` file.

Complete the following steps to configure trust anchors using an assembly tool.

Procedure

1. Configure an assembly tool to work with a Java Platform, Enterprise Edition (Java EE) enterprise application. For more information, see the related information on Assembly Tools.
2. Create a web services-enabled Java EE enterprise application.
3. Configure the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` bindings extensions file.
   a. Use an assembly tool to import your Java EE application.
   b. Click **Window** > **Open Perspective** > **Other** > **J2EE**.
   c. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**
   d. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab. The Client Deployment Descriptor is displayed.
   e. Locate the Port qualified name binding section and either select an existing entry or click **Add**, to add a new port binding. The web services client port binding editor displays for the selected port.
   f. Locate the Trust anchor section and click **Add**. The Trust anchor window is displayed.
      1) Enter a unique name within the port binding for the **Trust anchor name**.
         The name is used to reference the trust anchor that is defined.
      2) Enter the keystore password, path, and keystore type.
         The supported keystore types are the Java Cryptography Extension (JCE) and Java Cryptography Extension Keystores (JCEKS) types.

      Click **Edit** to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor.

When you start the application, the configuration is validated in the run time while the binding information is loading.

g. Save the changes.

4. Configure the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` bindings extensions file.
   a. Click **Window** > **Open perspective** > **J2EE**.
   b. Select the web services enabled Enterprise JavaBeans (EJB) or web module.
   c. In the Package Explorer window, click the `META-INF` directory for an EJB module or the `WEB-INF` directory for a web module.
   d. Right-click the `webservices.xml` file, select **Open with** > **Web services editor**, and click the bindings tab. The web services bindings editor is displayed.
   e. Locate the web service description bindings section and either select an existing entry or click **Add** to add a new web services descriptor.
   f. Click **Binding configurations**. The web services binding configurations editor is displayed for the selected web services descriptor.
   g. Locate the Trust anchor section and click **Add**. The Trust anchor dialog box is displayed.
      1) Enter a unique name within the binding for the **Trust anchor name**.

         This unique name is used to reference the trust anchor defined.

      2) Enter the keystore password, path, and keystore type. The supported keystore types are `JCE` and `JCEKS`.

      Click **Edit** to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor.

      When you start the application, the configuration is validated in the run time while the binding information is loading.

   h. Save the changes.

Results

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the web services is acting as client) to verify the signer certificate.

Example

The request receiver or the response receiver (if the web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

What to do next

To complete the signing information configuration process for request receiver, complete the following tasks:

1. “Configuring the server for request digital signature verification: Verifying the message parts” on page 1697
2. “Configuring the server for request digital signature verification: choosing the verification method” on page 1698

To complete the process for the response receiver, if the web services is acting as a client, complete the following tasks:

1. “Configuring the client for response digital signature verification: verifying the message parts” on page 1704
2. “Configuring the client for response digital signature verification: choosing the verification method” on page 1706

Configuring the client-side collection certificate store using an assembly tool

You can configure the client-side collection certificate store using the assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using an assembly tool or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the assembly tool.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab, which is located at the bottom of deployment descriptor editor within the assembly tool. The Client Deployment Descriptor is displayed.
5. Click the **Port binding** tab in deployment descriptor editor within the assembly tool. The web services client port binding window is displayed.
6. Select one of the port-qualified name binding entries.
7. Expand the **Security response receiver binding configuration** > **certificate store list** > **Collection certificate store** section.
8. Click **Add** to create a new collection certificate store, click **Edit** to edit an existing certificate store, or click **Remove** to delete an existing certificate store.
9. Enter a name in the **Name** field. This name is referenced in the Certificate store reference field in the Signing info dialog box.
10. Leave the Provider field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you finish adding paths.

Configuring the server-side collection certificate store using an assembly tool

A *collection certificate store* is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collections of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. You can configure the server-side collection certificate store by using an assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure the collection certificate either by using an assembly tool or by using the WebSphere Application Server administrative console. Complete the following steps to configure the server-side collection certificate store using an assembly tool.

Procedure

1. Start an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, select **Open with** > **Web Services Editor**.
5. Click the Binding configurations tab in the web services editor within the assembly tool. The Web Service Binding Configuration window is displayed.
6. Select one of the web service description binding entries under the Port Component Binding section.
7. Expand the **Request receiver binding configuration details** > **Certificate store list** > **Collection certificate store** section.
8. Click **Add** to create a new collection certificate store, click **Edit** to edit an existing certificate store, or click **Remove** to delete an existing certification store.
9. Enter a name in the **Name** field. This name is referenced in the **Certificate store reference** field in the Signing info dialog.
10. Leave the **Provider** field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT]/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you finish adding paths.

Configuring key locators using an assembly tool

The following information provides instructions on how to configure key locators using an assembly tool.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure key locators in various locations within the assembly tool. The following procedure provides instructions on how to configure key locators at any of these locations because the concept is the same.

Procedure

1. Start an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab. The Client Deployment Descriptor is displayed.
5. Click the **WS Binding** tab in deployment descriptor editor within the assembly tool or the **Binding configurations** tab in the Web services editor within the assembly tool.
6. Expand one of the **Binding configuration** sections.
7. Expand the **Key locators** section.

8. Click **Add** to create a new key locator, click **Edit** to edit an existing key locator, or click **Remove** to delete an existing key locator.
9. Enter a key locator name. The name entered for the **Key locator name** is used to refer to the key locator from the Encryption information and Signing Information sections.
10. Enter a key locator class. The key locator class is the implementation of the `KeyLocator` interface. When using default implementations, select a class from the menu.
11. Determine whether to click **Use key store**. Select this option when you use the default implementations as they use key stores. If you click **Use key store**, complete the following steps:
    a. Enter a value in the key store storepass field. The key store storepass is the password used to access the key store.
    b. Enter a path name in the key store path field. The key store path is the location on the file system where the key store resides. Make sure that the location can be found wherever you deploy the application.
    c. Enter a type value in the key store type field. The valid types to enter are JKS and JCEKS. JKS is used when you are not using the Java Cryptography Extensions (JCE) policy. JCEKS is used when you are using JCE. Although the JCEKS type is more secure, it might decrease performance.
    d. Click **Add** to create an entry for a key in the key store.
        1) Enter a value in the Alias field.

           The key alias is a reference to this particular key from the Signing Information section.
        2) Enter a value in the Key pass field.

           The key pass is the password associated with the certificate which is created using the Java SE Development Kit 6 `keytool.exe` file.
        3) Enter a value in the Key name field.

           The key name refers to the alias of the certificate as found in the key store.
12. Click **Add** to create a custom property. The property can be used by custom key locator implementations. For example, you can use properties with the `WSIdKeyStoreMapKeyLocator` default implementation. The key locator implementation has the following property names:
    - *id_*, which maps to a credential user ID.
    - *mappedName_* , which maps to the key alias to use for this user name.
    - *default*, which maps to a key alias to use when a credential does not have an associated *id_* entry.

    A typical set of properties for this key locator might be: `id_1=user1`, `mappedName_1=key1`, `id_2=user2`, `mappedName_2=key2`, `default=key3`. If `user1` or `user2` authenticates, then the associated `key1` or `key2` is used, respectively. However, if none of the user properties authenticate or the user is not `user1` or `user2`, then `key3` is used.
    a. Enter a name in the Name field. The name entered is the property name.
    b. Enter a value in the Value field. This value entered is the property value.

Securing web services for Version 5.x applications using XML digital signature

XML digital signature is one of the methods WebSphere Application Server provides to secure your web services. It provides message integrity and authentication capabilities when used with SOAP messages.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services; XML digital signature is one of these methods. You can secure your web services by using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

XML digital signature provides both message integrity and authentication capabilities when it is used with SOAP messages. A message receiver can verify that attackers or accidents have not altered parts of the message after the message was signed by a key. If a message has a digital certificate issued by a certificate authority (CA) and a signature in the message is validated successfully by a public key in the certificate, it is proof that the signer has the corresponding private key. To use XML digital signature to secure web services, complete the following steps:

Procedure

1. Define the security constraints or extensions. To configure the security constraints, you must use an assembly tool. For more information, see the related information on Assembly Tools.
   a. Configure the client to digitally sign a message request. To configure the client, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The client in these steps is the request sender.
      1) Specify the message parts by following the steps found in “Configuring the client for request signing: digitally signing message parts” on page 1693.
      2) Select the method used to digitally sign the request message. You can select the digital signature method by following the steps in “Configuring the client for request signing: choosing the digital signature method” on page 1695.
   b. Configure the server to verify the digital signature that is used in the message request. To configure the server, you must specify which parts of the SOAP message, sent by the request sender, contain digitally signed information and which method was used to digitally sign the message. The settings chosen for the request receiver, or the server in this step, must match the settings chosen for the request sender in the previous step.
      1) Define the message parts by following the steps found in “Configuring the server for request digital signature verification: Verifying the message parts” on page 1697.
      2) Select the same method used by the request sender to digitally sign the message. You can select the digital signature method by following the steps in “Configuring the server for request digital signature verification: choosing the verification method” on page 1698
   c. Configure the server to digitally sign a message response. To configure the server, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The sender in these steps is the response sender.
      1) Specify which message parts to digitally sign by following the steps found in “Configuring the server for response signing: digitally signing message parts” on page 1701.
      2) Select the method used to digitally sign the response message. You can select the digital signature method by following the steps in “Configuring the server for response signing: choosing the digital signature method” on page 1703
   d. Configure the client to verify the digital signature that is used in the message response. To configure the client, you must specify which parts of the SOAP message sent by the response sender contain digitally signed information and which method was used to digitally sign the

message. The settings chosen for the response receiver, or client in this step, must match the settings chosen for the response sender in the previous step.

1) Define the message parts by following the steps found in “Configuring the client for response digital signature verification: verifying the message parts” on page 1704
2) Select the same method used by the response sender to digitally sign the message. You can select the digital signature method by following the steps in “Configuring the client for response digital signature verification: choosing the verification method” on page 1706

2. Define the client security bindings. To configure the client security bindings, complete the steps in either of the following topics:
   - “Configuring the client security bindings using an assembly tool” on page 1708
   - Configuring the security bindings on a server acting as a client using the administrative console
3. Define the server security bindings. To configure the server security bindings, complete the steps in either of the following topics:
   - “Configuring the server security bindings using an assembly tool” on page 1711
   - Configuring the server security bindings using the administrative console

Results

After completing these steps, you have secured your web services using XML digital signature.

Configuring the client for request signing: digitally signing message parts

To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within an assembly tool.

- “Configuring the client security bindings using an assembly tool” on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Extension** tab. The Client Deployment Descriptor is displayed.
5. Expand **Request sender configuration** > **Integrity**. *Integrity* refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet.

6. Indicate which parts of the message to sign by clicking **Add** and selecting **body**, **timestamp**, or **SecurityToken**. The following list contains descriptions of the message parts

   body The body is the user data portion of the message.

   timestamp
   The time stamp determines if the message is valid based on the time that the message is sent and then received. If **timestamp** is selected, proceed to the next step and select **Add created time stamp** to add a time stamp to a message.

   SecurityToken
   The security token authenticates the client. If this option is selected, the message is signed.

   You can choose to digitally sign the message using a time stamp if **Add created time stamp** is selected and configured. You can digitally sign the message using a security token if a login configuration authentication method is selected.
7. Optional: Expand the **Add created time stamp** section and select this option if you want a time stamp added to the message. You can specify an expiration time for the time stamp, which helps defend against replay attacks. The lexical representation for duration is the [ISO 8601] extended format *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years
   - *nM* represents the number of months
   - *nD* represents the number of days
   - *T* is the date and time separator
   - *nH* represents the number of hours
   - *nM* represents the number of minutes
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, the format is: `P1Y2M3DT10H30M`. Typically, you configure a message time stamp for about 10 to 30 minutes, for example, 10 minutes is represented as: `P0Y0M0DT0H10M0S`. The *P* character precedes time and date values.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the **Actor** fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **Actor** fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as

a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server **Actor** fields are synchronized.

What to do next

After you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message. See "Configuring the client for request signing: choosing the digital signature method" for more information.

Configuring the client for request signing: choosing the digital signature method

To configure the client for request signing, specify which message parts to digitally sign when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Security extensions** tab and the **Port binding** tab in the web services client editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 1708
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively. You must specify which parts of the message sent by the client must be digitally signed. See "Configuring the client for request signing: digitally signing message parts" on page 1693 for more information.

About this task

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment Descriptor Editor**, and click the **WS Binding** tab. The Client Deployment Descriptor is displayed.
5. Expand **Security request sender binding configuration** > **Signing information**.
6. Select **Edit** to view the signing information and select a digital signature method from the **Signature method algorithm** field. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following website http://www.w3.org/TR/xmldsig-core.

Table 235. Digital signature methods. The digital signature method information is stored in the client deployment descriptor.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the <SignedInfo> element before the information is digested as part of the signature operation.
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the <DigestValue> element. Signing the <DigestValue> element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signature method algorithm	Converts the canonicalized <SignedInfo> element into the <SignatureValue> element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signing key name	Represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is found in the key store and is used to sign the request.
Signing key locator	Represents a reference to a key locator implementation class that locates the correct hey store where the alias and the certificate exist.

7. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the **Digest method algorithm** and **Signature method algorithm** drop-down lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the WebSphere administrative console.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the web services client editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the web services editor within an assembly tool:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **Actor** fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method is used to digitally sign a message when the client sends a message to a server.

What to do next

After you configure the client to digitally sign the message, you must configure the server to verify the digital signature. See “Configuring the server for request digital signature verification: Verifying the message parts” for more information.

Configuring the server for request digital signature verification: Verifying the message parts

Configure the server for request digital signature verification by modifying the extensions to indicate which parts of the request to verify.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Extensions** tab and the **Binding Configurations** tab in the web services editor within the assembly tools:

- “Configuring the server security bindings using an assembly tool” on page 1711
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and the Web Services Security bindings, respectively. Also, you must specify which parts of the message sent by the client must be digitally signed. See “Configuring the client for request signing: digitally signing message parts” on page 1693 to determine which message parts are digitally signed. The message parts specified for the client request sender must match the message parts specified for the server request receiver.

About this task

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which parts of the request to verify.

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Extensions tab in the web services editor.
6. Expand the **Request receiver service configuration details** > **Required integrity** section. Required integrity refers to the parts of the message that require digital signature verification. The purpose of digital signature verification is to make sure that the message parts have not been modified while transmitting across the Internet.
7. Indicate parts of the message to verify by clicking **Add**, and selecting one of the following three parts: **body**, **Timestamp**, or **SecurityToken**. You can determine which parts of the message to verify by looking at the web service request sender configuration in the client application. To view the web service request sender configuration information in the web services client editor, click the Security extensions tab and expand **Request sender configuration** > **Integrity**. The following includes a list and description of the message parts:

   Body This is the user data portion of the message.

Timestamp
The time stamp determines if the message is valid based on the time that the message is sent and then received. If **Timestamp** is selected, proceed to the next step to `Add Created Time Stamp` to the message.

SecurityToken
The security token authenticates the client. If **SecurityToken** is selected, the message is signed.

8. Optional: Expand the **Add received time stamp** section. The Add Received Time Stamp value indicates to validate the Add Created Time Stamp option configured by the client. You must select this option if you selected the Add Created Time Stamp on the client. The time stamp ensures message integrity by indicating the timeliness of the request. This option helps defend against replay attacks.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the server when the client sends a message to a server.

What to do next

After you specify which message parts contain a digital signature that must be verified by the server, you must configure the server to recognize the digital signature method used to digitally sign the message. See “Configuring the server for request digital signature verification: choosing the verification method” for more information.

Configuring the server for request digital signature verification: choosing the verification method

To configure the server for request digital signature verification, use an assembly tool to modify the extensions and indicate which digital signature method the server will use during verification.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the Extensions tab and the Binding Configurations tab in the Web Services Editor within the IBM assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 1711
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. You must specify which message parts contain digital signature information that must be verified by the server. See "Configuring the server for request digital signature verification: Verifying the message parts" on page 1697. The message parts specified for the client request sender must match the message parts specified for the server request receiver. Likewise, the digital signature method chosen for the client must match the digital signature method used by the server.

About this task

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the server will use during verification.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab.
6. Expand the **Security request receiver binding configuration details** > **Signing information** section.
7. Click **Edit** to edit the signing information. The signing information dialog is displayed, select or enter the following information:
   - Canonicalization method algorithm
   - Digest method algorithm
   - Signature method algorithm
   - Use certificate path reference
   - Trust anchor reference
   - Certificate store reference
   - Trust any certificate

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which is located at the following web address: http://www.w3.org/TR/xmldsig-core.

Table 236. Digital signature methods. The digital signature method is part of the binding configuration.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the `<SignedInfo>` element before it is digested as part of the signature operation. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.

Table 236. Digital signature methods (continued). The digital signature method is part of the binding configuration.

Name	Purpose
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the <DigestValue> element. The signing of the <DigestValue> element binds resource content to the signer key. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Signature method algorithm	Converts the canonicalized <SignedInfo> element into the <SignatureValue> element. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Use certificate path reference or **Trust any certificate**	Validates a certificate or signature sent with a message. When a message is signed, the public key used to sign it is sent with the message. This public key or certificate might not be validated at the receiving end. By selecting **User certificate path reference**, you must configure a trust anchor reference and a certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	Refers to a key store that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	Contains a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Signature method algorithm and Digest method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the Actor URI field.
- Click **Security extensions** > **Request sender configuration > Details** and indicate the actor information in the Actor field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the Actor field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified the method that the server uses to verify the digital signature in the message parts.

What to do next

After you configure the client for request signing and the server for request digital signature verification, you must configure the server and the client to handle the response. Next, specify the response signing for the server. See “Configuring the server for response signing: digitally signing message parts” for more information.

Configuring the server for response signing: digitally signing message parts

Use an assembly tool to specify which message parts to digitally sign when configuring the server for response signing.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this topic supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Extensions** tab and the **Binding** configurations tab in the web services editor within the IBM assembly tools:

- “Configuring the server security bindings using an assembly tool” on page 1711
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which message parts to digitally sign when configuring the server for response signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application _name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the Web Services Editor within the assembly tool.
6. Expand **Response sender service configuration details** > **Integrity**. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see XML digital signature.
7. Indicate the parts of the message to sign by clicking **Add**, and selecting **Body**, **Timestamp**, or **SecurityToken**.

   The following list contains descriptions of the message parts:

   Body The body is the user data portion of the message.

   Timestamp
   The time stamp determines if the message is valid based on the time that the message is sent and then received. If this option if selected, proceed to the next step and click **Add Created Time Stamp**, which indicates that the time stamp is added to the message.

SecurityToken
The security token is used for authentication. If this option is selected, the authentication information is added to the message.

8. Optional: Expand the **Add created time stamp** section. Select this option if you want a time stamp added to the message. You can specify an expiration time for the time stamp, which helps defend against replay attacks. The lexical representation for duration is the ISO 8601 extended format, *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years.
   - *nM* represents the number of months.
   - *nD* represents the number of days.
   - *T* is the date and time separator.
   - *nH* represents the number of hours.
   - *nM* represents the number of minutes.
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, the format is: `P1Y2M3DT10H30M`. Typically, you configure a message time stamp for about 10 to 30 minutes. 10 minutes is represented as: `P0Y0M0DT0H10M0S`. The *P* character precedes time and date values.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations:
- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the **Actor URI** field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations:
- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts to digitally sign when the server sends a response to the client.

What to do next

After you specifying which message parts to digitally sign, you must specify which method is used to digitally sign the message. See “Configuring the server for response signing: choosing the digital signature method” on page 1703 for more information.

Configuring the server for response signing: choosing the digital signature method

Use an assembly tool to specify which digital signature method to use when configuring the server for response signing.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the Extensions tab and the **Binding configurations** tab in the web services editor within the IBM assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 1711
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which digital signature method to use when configuring the server for response signing:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application _name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab.
6. Expand **Response sender binding configuration details** > **Signing information**.
7. Click **Edit** to choose a signing method. The signing info dialog is displayed and either select or enter the following information:
   - Canonicalization method algorithm
   - Digest method algorithm
   - Signature method algorithm
   - Signing key name
   - Signing key locator

   The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core.

Table 237. Digital signature methods. Use the methods to configure the server for response signing.

Name	Purpose
Canonicalization method algorithm	Canonicalizes the <SignedInfo> element before the information is digested as part of the signature operation. Use the same algorithm on the client response receiver. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

Table 237. Digital signature methods (continued). Use the methods to configure the server for response signing.

Name	Purpose
Digest method algorithm	Applies to the data after transforms are applied, if specified, to yield the <DigestValue> element. Signing the <DigestValue> element binds resource content to the signer key. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signature method algorithm	Converts the canonicalized <SignedInfo> element into the <SignatureValue> element. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signing key name	Represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is found in the key store and is used to sign the request.
Signing key locator	Represents a reference to a key locator implementation class that locates the correct key store where the alias and certificate exists. For more information on configuring key locators, see the following file: • "Configuring key locators using an assembly tool" on page 1690

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Signature method algorithm and Digest method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

You have specified which method is used to digitally sign a message when the server sends a message to a client.

What to do next

After you configure the server to digitally sign the response message, you must configure the client to verify the digital signature contained in the response message. See "Configuring the client for response digital signature verification: verifying the message parts" for more information.

Configuring the client for response digital signature verification: verifying the message parts

To configure the Web Services Security extensions and the Web Services Security bindings, use the **WS Extension** tab and the **WS Binding** tab in the Client Deployment Descriptor within an assembly tool.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to becomes familiar with the **WS Extension** tab and the **WS Binding** tab in the Client Deployment Descriptor within the assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

You can use these two tabs to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which parts of the response to verify.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file and click **Open With** > **Deployment descriptor editor**.
5. Click the **WS extension** tab.
6. Expand the **Response receiver configuration** > **Required integrity** section. Required integrity refers to parts that require digital signature verification. Digital signature verification decreases the risk that the message parts have been modified while the message is transmitted across the Internet.
7. Indicate the parts of the message that must be verified. You can determine which parts of the message to verify by looking at the web service response sender configuration. Click **Add** and select one of the following parts:

   Body The body is the user data portion of the message.

   Timestamp
   The time stamp determines if the message is valid based on the time that the message is sent and then received. If the timestamp option is selected, proceed to the next step to add a received time stamp to the message.

   Securitytoken
   The security token authenticates the client. If the Securitytoken option is selected, the message is signed.
8. Optional: Expand the **Add received time stamp** section. Select **Add received time stamp** to add the received time stamp to the message.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the web services client editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the Actor URI field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the Actor field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the web services editor within an assembly tool:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the Actor field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure

that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the client when the server sends a response message to the client.

What to do next

After you specify which message parts contain a digital signature that must be verified by the client, you must configure the client to recognize the digital signature method used to digitally sign the message. See "Configuring the client for response digital signature verification: choosing the verification method" for more information.

Configuring the client for response digital signature verification: choosing the verification method

You can configure the Web Services Security extensions and Web Services Security bindings using the **WS extension** tab and the **WS binding** tab in the web services editor within an assembly tool.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS extension** tab and the **WS binding** tab in the web services editor within the IBM assembly tools:

- "Configuring the server security bindings using an assembly tool" on page 1711
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which message parts contain digital signature information that must be verified by the client. See "Configuring the client for response digital signature verification: verifying the message parts" on page 1704 to specify which message parts are digitally signed by the server and must be verified by the client. The message parts specified for the server response sender must match the message parts specified for the client response receiver. Likewise, the digital signature method chosen for the server must match the digital signature method used by the client.

About this task

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the client will use during verification.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Binding** tab.

6. Expand the **Security response receiver binding configuration** > **Signing information** section.
7. Click **Edit** to select a digital signature method. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Signing key name**
   - **Signing key locator**

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which can be found at: http://www.w3.org/TR/xmldsig-core.

Table 238. Digital signature methods. Use the methods to configure the client for response digital signature verification.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the `<DigestValue>` binds resource content to the signer key. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is transmitted with the message. To validate this public key at the receiving end, configure a certificate path reference. By selecting **User certificate path reference,** you must configure a trust anchor reference and certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a keystore that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Signature method algorithm and Digest method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when running the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the web services client editor within an assembly tool:

- Click **Security extensions** > **Client service configuration details** and indicate the actor information in the Actor URI field.
- Click **Security extensions** > **Request sender configuration** > **Details** and indicate the actor information in the Actor field.

You must configure the same actor strings for the web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the web services editor within an assembly tool:

- Click **Security extensions** > **Server service configuration**.
- Click **Security extensions** > **Response sender service configuration details** > **Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The actor fields might be different when you have web services acting as a gateway to other web services. However, in all other cases, make sure that the actor information matches on the client and server. When web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, web services do not process the message from a client. Instead, these web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method the client uses to verify the digital signature in the message parts.

What to do next

After you configure the server for response signing and the client for request digital signature verification, verify that you have configured the client and the server to handle the message request.

Configuring the client security bindings using an assembly tool

Use the web services client editor within an assembly tool to include the binding information, that describes how to run the security specifications found in the extensions, in the client enterprise archive (EAR) file.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

When configuring a client for Web Services Security, the bindings describe how to run the security specifications found in the extensions. Use the web services client editor within an assembly tool to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a web service or from a web service accessing a downstream web service. This document focuses on the pure client situation. However, the concepts, and in most cases the steps, also apply when a web service is configured to communicate downstream to another web service that has client bindings. Complete the following steps to edit the security bindings on a pure client (or server acting as a client) using an assembly tool:

Procedure

1. Import the web services client EAR file into an assembly tool. When you edit the client bindings on a server acting as a client, the same basic steps apply. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.

4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**. The Client Deployment Descriptor is displayed.
5. Click the **WS Extension** tab and select the port QName bindings that you want to configure. The Web Services Security extensions are configured for outbound requests and inbound responses. You need to configure the following information for Web Services Security extensions. These topics are discussed in more detail in other sections of the documentation.

   Request sender configuration details

   Details
   "Configuring the client for request signing: digitally signing message parts" on page 1693

   Integrity
   "Configuring the client for request signing: digitally signing message parts" on page 1693

   Confidentiality
   "Configuring the client for request encryption: Encrypting the message parts" on page 1715

   Login Config

   BasicAuth
   "Configuring the client for basic authentication: specifying the method" on page 1726

   IDAssertion
   "Configuring the client for identity assertion: specifying the method" on page 1734

   Signature
   "Configuring the client for signature authentication: specifying the method" on page 1741

   LTPA "Configuring the client for LTPA token authentication: specifying LTPA token authentication" on page 1750

   ID assertion
   "Configuring the client for identity assertion: specifying the method" on page 1734

   Add created time stamp
   "Configuring the client for request signing: digitally signing message parts" on page 1693

   Response receiver configuration details

   Required integrity
   "Configuring the client for response digital signature verification: verifying the message parts" on page 1704

   Required confidentiality
   "Configuring the client for response decryption: decrypting the message parts" on page 1722

   Add received time stamp
   "Configuring the client for response digital signature verification: verifying the message parts" on page 1704
6. Click the **WS binding** tab and select the port qualified name binding that you want to configure. The Web Services Security bindings are configured for outbound requests and inbound responses. You need to configure the following information for Web Services Security bindings. These topics are discussed in more details in other sections of the documentation.

   Security request sender binding configuration

   Signing information
   "Configuring the client for request signing: choosing the digital signature method" on page 1695

   Encryption information
   "Configuring the client for request encryption: choosing the encryption method" on page 1716

Key locators
"Configuring key locators using an assembly tool" on page 1690

Login binding

BasicAuth
"Configuring the client for basic authentication: collecting the authentication information" on page 1728

ID assertion
"Configuring the client for identity assertion: collecting the authentication method" on page 1735

Signature
"Configuring the client for signature authentication: collecting the authentication information" on page 1742

LTPA "Configuring the client for LTPA token authentication: collecting the authentication method information" on page 1751

Security response receiver binding configuration

Signing information
"Configuring the client for response digital signature verification: choosing the verification method" on page 1706

Encryption information
"Configuring the client for response decryption: choosing a decryption method" on page 1723

Trust anchor
"Configuring trust anchors using an assembly tool" on page 1687

Certificate store list
"Configuring the client-side collection certificate store using an assembly tool" on page 1689

Key locators
"Configuring key locators using an assembly tool" on page 1690

What to do next

Important: When configuring the security request sender binding configuration, you must synchronize the information used to perform the specified security with the security request receiver binding configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects because there is no negotiation during run time to determine the requirements of the server.

For example, when configuring the encryption information in the security request sender binding Configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This example illustrates the important relationship between the client and server configuration. Additionally, when configuring the security response receiver binding configuration, the server must send the response using security information known by this client security response receiver binding configuration.

The following table shows the related configurations between the client and the server. The client request sender and the server request receiver are relative configurations that must be synchronized with each other. The server response sender and the client response receiver are related configurations that must be synchronized with each other. Note that the related configurations are end points for any request or response. One end point must communicate its actions with the other end point because run time requirements are not negotiated.

Table 239. Related configurations. The configurations must be synchronized with each other.

Client configuration	Server configuration
Request sender	Request receiver
Response receiver	Response sender

Configuring the server security bindings using an assembly tool

Use an assembly tool to edit bindings for a web service after these bindings are deployed on a server.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to importing the web services enterprise archive (EAR) file into the assembly tool, make sure that you have already run the **WSDL2Java** command on your web service to enable your Java Platform, Enterprise Edition (Java EE) application. You must import the Web services EAR file into the assembly tool.

About this task

Create an Enterprise JavaBeans (EJB) file Java archive (JAR) file or a web application archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the **WSDL2Java** command. For more information, read about the **WSDL2Java** command for JAX-RPC applications. You can edit these files using the web services editor in the assembly tool.

When configuring server-side security for Web Services Security, the security extensions configuration specifies what security is performed, the security bindings configuration indicates how to perform what is specified in the security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference these elements from the WAR and JAR binding configurations.

Open the web services editor in an assembly tool to begin editing the server security extensions and bindings. The following steps can locate the server security extensions and bindings. Other tasks specify how to configure each section of the extensions and bindings in more detail.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java EE perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Configure the server for inbound requests and outbound responses security configuration. To configure the server for inbound requests and outbound responses, complete the following steps:
   a. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
   b. Right-click the `webservices.xml` file and click **Open with** > **Web services editor**. The `webservices.xml` file represents the server-side (inbound) web services configuration. The `webservicesclient.xml` file represents the client-side (outbound) web services configuration.
4. In the web services editor (for the `webservices.xml` file and inbound requests and outbound responses web services configuration), there are several tabs at the bottom of the editor including Web Services,

Port Components, Handlers, Security Extensions, Bindings, and Binding Configurations. The security extensions are edited using the **Security Extensions** tab. The security bindings are edited using the Security Bindings tab.

a. Click the **WS Extensions** tab and select the port component binding to edit. The Web Services Security extensions are configured for inbound requests and outbound responses. You need to configure the following information for Web Services Security extensions. These topics are discussed in more detail in other topics in the documentation.

   Request receiver service configuration details

   Required integrity
   "Configuring the server for request digital signature verification: Verifying the message parts" on page 1697

   Required confidentiality
   "Configuring the server for request decryption: decrypting the message parts" on page 1717

   Login config

   BasicAuth
   "Configuring the server to handle basic authentication information" on page 1731

   ID assertion
   "Configuring the server to handle identity assertion authentication" on page 1736

   Signature
   "Configuring the server to support signature authentication" on page 1744

   LTPA "Configuring the server to handle LTPA token authentication information" on page 1752

   Add received time stamp
   "Configuring the server for request digital signature verification: Verifying the message parts" on page 1697

   Response sender service configuration details

   Details
   "Configuring the server for response signing: digitally signing message parts" on page 1701

   Integrity
   "Configuring the server for response signing: digitally signing message parts" on page 1701

   Confidentiality
   "Configuring the server for response encryption: encrypting the message parts" on page 1720

   Add created time stamp
   "Configuring the server for response signing: digitally signing message parts" on page 1701

b. Click the **Binding Configurations** tab and select the port component binding to edit. The Web Services Security bindings are configured for inbound requests and outbound responses. You need to configure the following information for Web Services Security bindings. These topics are discussed in more details in other topics in the documentation.

   Response receiver binding configuration details

   Signing Information
   "Configuring the server for request digital signature verification: choosing the verification method" on page 1698

   Encryption Information
   "Configuring the server for request decryption: choosing the decryption method" on page 1718

Trust Anchor
"Configuring trust anchors using an assembly tool" on page 1687

Certificate Store List
"Configuring the server-side collection certificate store using an assembly tool" on page 1689

Key Locators
"Configuring key locators using an assembly tool" on page 1690

Login Mapping

Basic auth
"Configuring the server to validate basic authentication information" on page 1732

ID assertion
"Configuring the server to validate identity assertion authentication information" on page 1738

Signature
"Configuring the server to validate signature authentication information" on page 1745

LTPA "Configuring the server to validate LTPA token authentication information" on page 1753

Trusted ID evaluator

Trusted ID evaluator reference

Response sender binding configuration details

Signing information
"Configuring the server for response signing: choosing the digital signature method" on page 1703

Encryption information
"Configuring the server for response encryption: choosing the encryption method" on page 1721

Key locators
"Configuring key locators using an assembly tool" on page 1690

What to do next

Configure the client for outbound requests and inbound responses security configuration by right-clicking the `webservicesclient.xml` file and clicking **Open With** > **Deployment descriptor editor**. For more information, see "Configuring the client security bindings using an assembly tool" on page 1708.

Configuring XML encryption for Version 5.x web services with an assembly tool

XML encryption is one method that WebSphere® Application Server provides to secure your web services. It enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document.

Securing web services for Version 5.x applications using XML encryption

XML encryption is one method that WebSphere Application Server provides to secure your web services. It enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. XML encryption is one of these methods. You can secure your web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

XML encryption enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as an XML document. Like XML digital signature, a message is sent by the client as the request sender to the server as the request receiver. The response is sent by the server as the response sender to the client as the request receiver. Unlike XML digital signature, which verifies the authenticity of the sender, XML encryption scrambles the message content using a key, which can be unscrambled by a receiver that possesses the same key. You can use XML encryption in conjunction with XML digital signature to scramble the content while verifying the authenticity of the message sender.

To use XML encryption to secure web services, you must use an assembly tool. For more information, see the related information on Assembly Tools.

To securing web services for Version 5.*x* applications using XML encryption, complete the following steps:

Procedure

1. Specify the encryption settings for the request sender. The message parts and the encryption method settings chosen for the request sender on the client must match the message parts and the method settings chosen for the request receiver on the server. To specify the encryption settings for the request sender:
   a. “Configuring the client for request encryption: Encrypting the message parts” on page 1715.
   b. “Configuring the client for request encryption: choosing the encryption method” on page 1716.
2. Specify the encryption settings for the request receiver. The decryption settings chosen for the request receiver must match the encryption settings chosen for the request sender.

   To specify the decryption settings for the request receiver:
   a. “Configuring the server for request decryption: decrypting the message parts” on page 1717.
   b. “Configuring the server for request decryption: choosing the decryption method” on page 1718.
3. Specify the encryption settings for the response sender. The message parts and the encryption method settings chosen for the response sender on the server must match the message parts and the method settings chosen for the response receiver on the client. To specify the encryption settings for the response sender:
   a. “Configuring the server for response encryption: encrypting the message parts” on page 1720.
   b. “Configuring the server for response encryption: choosing the encryption method” on page 1721.
4. Specify the encryption settings for the response receiver.

Remember: The decryption settings chosen for the response receiver must match the encryption settings chosen for the response sender.

To specify the decryption settings for the response receiver, complete the following steps:

a. "Configuring the client for response decryption: decrypting the message parts" on page 1722.

b. "Configuring the client for response decryption: choosing a decryption method" on page 1723.

Results

After completing these steps, you have secured your web services using XML encryption.

Configuring the client for request encryption: Encrypting the message parts

To configure the client for request encryption, specify which message parts to encrypt when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to familiarize yourself with the **WS Extensions** tab and the **WS Binding** tab in the Client Deployment Descriptor Editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which message parts to encrypt when configuring the client for request encryption:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS extensions** tab, which is located at the bottom of Client Deployment Descriptor Editor within the assembly tool.
6. Expand **Request sender configuration** > **Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting , see XML encryption.
7. Select the parts of the message that you want to encrypt by clicking **Add**. You can select one of the following parts:

   Bodycontent
   User data portion of the message

Usernametoken
Basic authentication information, if selected

What to do next

After you specify which message parts to encrypt, you must specify which method to use to encrypt the request message. See “Configuring the client for request encryption: choosing the encryption method” for more information.

Configuring the client for request encryption: choosing the encryption method

To configure the client for request encryption, specify which encryption method to use when configuring the client.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to familiarize yourself with the **WS Extensions** tab and the **WS Binding** tab in the Client Deployment Descriptor editor within an assembly tool:

- “Configuring the client security bindings using an assembly tool” on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which encryption method to use when configuring the client for request encryption:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS binding** tab, which is located at the bottom of the Client Deployment Descriptor editor within the assembly tool.
6. Expand **Security request sender binding configuration** > **Encryption information**.
7. Select an encryption option and click **Edit** to view the encryption information or click **Add** to add another option. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Refers to the name of the encryption information entry.

   Data encryption method algorithm
   Encrypts and decrypts data in fixed size, multiple octet blocks.

Key encryption method algorithm
Represents public key encryption algorithms that are specified for encrypting and decrypting keys.

Encryption key name
Represents a Subject (**Owner** field of the certificate) from a public key certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

The key chosen must be a public key of the target. Encryption must be done using the public key and decryption must be done by the target using the private key (the personal certificate of the target).

Encryption key locator
Represents a reference to a key locator implementation class that locates the correct key store where the alias and the certificate exist. For more information on configuring key locators, see "Configuring key locators using an assembly tool" on page 1690 and Configuring key locators using the administrative console.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the **Data Encryption method algorithm** and **Key Encryption method algorithm** dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the WebSphere administrative console.

Results

For more information, see "Configuring key locators using an assembly tool" on page 1690 and Configuring key locators using the administrative console.

What to do next

You must specify which parts of the request message to encrypt. See "Configuring the client for request encryption: Encrypting the message parts" on page 1715 if you have not previously specified this information.

Configuring the server for request decryption: decrypting the message parts

Use the **WS Extensions** tab and the **WS Binding** configurations tab to specify which parts of the request message must be decrypted by the server.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.x and later applications.

Complete this task to specify which parts of the request message must be decrypted by the server. You must know which parts of the request message the client encrypts because the server must decrypt the same message parts.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Binding** configurations tab:

- "Configuring the server security bindings using an assembly tool" on page 1711
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to configure the request receiver extensions:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META_INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Required confidentiality** section.
7. Select the parts of the message to decrypt. The message parts selected for the request decryption on the server must match the message parts selected for the message encryption on the client. Click **Add** and select either of the following message parts:

   bodycontent
   The user data section of the message.

   usernametoken
   This token is the basic authentication information.

What to do next

After you specify which parts of the request message to decrypt, you must specify the method to use decrypt the message. See “Configuring the server for request decryption: choosing the decryption method” for more information.

Configuring the server for request decryption: choosing the decryption method

You can use an assembly tool and the administrative console to configure the Web Services Security extensions and Web Services Security bindings.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Bindings** tab:

- “Configuring the server security bindings using an assembly tool” on page 1711
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete this task to specify which decryption method is used by the server to decrypt the request message. You must know which decryption method the client uses because the server must use the same method.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META_INF**.
4. Right-click the `webservices.xml` file, select **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Encryption information** section.
7. Click **Edit** to view the encryption information. The following table describes the purpose for each of these selections. Some definitions are taken from the XML-Encryption specification , which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Represents the name of this encryption information entry; an alias for the entry.

   Data encryption method algorithm
   Encrypts and decrypts data in fixed size, multiple octet blocks. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Key encryption method algorithm
   Represents algorithms specified for encrypting and decrypting keys. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Encryption key name
   Represents a Subject from a personal certificate, which is typically a distinguished name (DN) that is found by the encryption key locator. The subject is used by the key encryption method algorithm to decrypt the secret key, and the secret key is used to decrypt the data.

   The key chosen must be a private key in the key store configured by the key locator. The key requires the same Subject used by the client to encrypt the data. Encryption must be done using the public key and decryption by using the private key (personal certificate). To ensure that the client encrypts the data with the correct public or private key, you must extract the public key from the server key store and add it to the key store specified in the encryption configuration information for the client request sender.

   For example, the personal certificate of a server is `CN=Bob, O=IBM, C=US`. Therefore the server contains the public and private key pair. The client sending the request should encrypt the data using the public key for `CN=Bob, O=IBM, C=US`. The server decrypts the data using the private key for `CN=Bob, O=IBM, C=US`.

   Encryption key locator
   Represents a reference to a key locator implementation class that finds the correct keystore where the alias and the certificate exist. For more information on configuring key locators, go to the following sections: “Configuring key locators using an assembly tool” on page 1690 and Configuring key locators using the administrative console.
8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Data Encryption method algorithm and Key Encryption method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application

Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

It is important to note that for decryption, the encryption key name chosen must refer to a personal certificate that can be located by the key locator of the server referenced in the encryption information. Enter the Subject of the personal certificate here, which is typically a Distinguished Name (DN). The Subject uses the default key locator to find the key. If a custom key locator is written, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that finds the correct key store where this alias and certificate exist. Refer to “Configuring key locators using an assembly tool” on page 1690 and Configuring key locators using the administrative console for more information.

What to do next

You must specify which parts of the request message to decrypt. See “Configuring the server for request decryption: decrypting the message parts” on page 1717 if you have not previously specified this information.

Configuring the server for response encryption: encrypting the message parts

You can specify which parts of the response message to encrypt when configuring the server for response encryption.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Bindings** tab in the Web services editor within an assembly tool:

- “Configuring the server security bindings using an assembly tool” on page 1711
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which parts of the response message to encrypt when configuring the server for response encryption:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window > Open Perspective > J2EE**.
3. Click **EJB Projects >** ***application_name*** **> ejbModule > META_INF**.
4. Right-click the `webservices.xml` file, select **Open with > Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the Web Services Editor within the assembly tool.

6. Expand **Response sender service configuration details > Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone understanding the message flowing across the Internet. With confidentiality specifications, the response is encrypted before it is sent and decrypted when it is received at the correct target.
7. Select the parts of the response that you want to encrypt by clicking **Add** and selecting **Bodytoken** or **Usernametoken**. The following information describes the message parts:

   Bodycontent
   User data portion of the message.

   Usernametoken
   Basic authentication information, if selected.

   A user name token does not appear in the response so you do not need to select this option for the response. If you select this option, make sure that you also select it for the client response receiver. If you do not select this option, make sure that you do not select it for the client response receiver.

What to do next

After you specify which message parts to encrypt, you must specify which method to use message encryption. See the task for choosing the encryption method when configuring the server for response encryption.

Configuring the server for response encryption: choosing the encryption method

You can specify which method the server uses to encrypt the response message.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **Extensions** tab and the **Binding configurations** tab in the web services editor within an assembly tool:

- "Configuring the server security bindings using an assembly tool" on page 1711
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which method the server uses to encrypt the response message:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META_INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configuration**s tab, which is located at the bottom of the Web Services Editor within the assembly tool.
6. Expand **Response sender binding configuration details** > **Encryption information**.

7. Click **Edit** to view the encryption information. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Refers to the name of the encryption information entry.

   Data encryption method algorithm
   Encrypts and decrypts data in fixed size, multiple octet blocks. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

   Key encryption method algorithm
   Represents public key encryption algorithms that are specified for encrypting and decrypting keys. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

   Encryption key name
   Represents a Subject from a public key certificate typically distinguished name (DN) that is found by the encryption key locator and used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

   The key name chosen in the server response sender encryption information must be the public key of the key configured in the client response receiver encryption information. Encryption by the response sender must be done using the public key and decryption must be done by the response receiver using the associated private key (the personal certificate of the response receiver).

   Encryption key locator
   The encryption key locator represents a reference to a key locator implementation class that finds the correct key store where the alias and the certificate exist. For more information, see the tasks for configuring key locators.
8. Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Data Encryption method algorithm and Key Encryption method algorithm drop-down lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

The encryption key name chosen must refer to a public key of the response receiver. For the encryption key name, use the Subject of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator , the encryption key name might be anything that is used by the key locator to find the correct encryption key (a public key). The encryption key locator references the implementation class that finds the correct key store where the alias and certificate exist.

What to do next

You must specify which parts of the response message to encrypt. See the task for configuring the server for response encryption if you have not previously specified this information.

Configuring the client for response decryption: decrypting the message parts

To configure the client for response decryption, specify which response message parts to decrypt when configuring the client. The server response encryption and client response decryption configurations must match.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Binding** tab in the Client Deployment Descriptor Editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and the Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which response message parts to decrypt when configuring the client for response decryption. The server response encryption and client response decryption configurations must match.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extensions** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Response receiver configuration** > **Required confidentiality** section.
7. Select the parts of the message that you must decrypt by clicking **Add** and selecting either **Bodycontent** or **Usernametoken**. The following information describes these message parts:

   Bodycontent
   The user data portion of the message.

   Usernametoken
   The basic authentication information, if selected.

   The information selected in this step is encrypted by the server in the response sender.

   Important: A Username Token is typically not sent in the response. Thus, you usually do not need to select username token.

What to do next

After you specify which message parts to decrypt, you must specify which method to use when decrypting the response message. See "Configuring the client for response decryption: choosing a decryption method" for more information.

Configuring the client for response decryption: choosing a decryption method

To configure the client for response decryption, specify which decryption method to use when the client decrypts the response message. The server response encryption and client response decryption configurations must match.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Prior to completing these steps, read either of the following topics to become familiar with the **WS Extensions** tab and the **WS Bindings** tab in the Client Deployment Descriptor Editor within an assembly tool:

- "Configuring the client security bindings using an assembly tool" on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

About this task

Complete the following steps to specify which decryption method to use when the client decrypts the response message. The server response encryption and client response decryption configurations must match.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Binding** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Security response receiver binding configuration** > **Encryption information** section. For more information on encrypting and decrypting SOAP messages, see XML encryption.
7. Click **Edit** to view the encryption information. The following table describes the purpose for this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following web address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Refers to the alias that is used for the encryption information entry.

   Data encryption method algorithm
   Encrypts and decrypts data in fixed size, multiple octet blocks.

   Key encryption method algorithm
   Represents public key encryption algorithms specified for encrypting and decrypting keys.

   Encryption key name
   Represents a Subject from a personal certificate, which is typically a distinguished name (DN) that is found by the encryption key locator. The Subject is used by the key encryption method algorithm to decrypt the secret key. The secret key is used to decrypt the data.

   Important: The key chosen must be a private key of the client. Encryption must be done using the public key and decryption must be done by the private key (personal certificate). For example, the personal certificate of the client is: `CN=Alice, O=IBM, C=US`. Therefore, the client contains the public and private key pair. The target

server that sends the response encrypts the secret key by using the public key for `CN=Alice, O=IBM, C=US`. The client decrypts the secret key by using the private key for `CN=Alice, O=IBM, C=US`.

Encryption key locator
Represents a reference to a key locator implementation class that finds the correct key store where the alias and the certificate exist. For more information on configuring key locators, see "Configuring key locators using an assembly tool" on page 1690 and Configuring key locators using the administrative console.

8. Optional: Select **Show only FIPS Compliant Algorithms** if you only want the FIPS compliant algorithms to be shown in the Data Encryption method algorithm and Key Encryption method algorithm dropdown lists. Use this option if you expect this application to be run on a WebSphere Application Server that has set the **Use the United States Federal Information Processing Standard (FIPS) algorithms** option in the SSL certificate and key management panel of the administrative console for WebSphere Application Server.

Results

For decryption, the encryption key name chosen must refer to a personal certificate that can be located by the client key locator. The Subject (**owner** field of the certificate) of the personal certificate should be entered in the Encryption key name, this is typically a Distinguished Name (DN). The default key locator uses the Encryption key name to find the key within the keystore. If you write a custom key locator, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information, see "Configuring key locators using an assembly tool" on page 1690 and Configuring key locators using the administrative console.

What to do next

You must specify which parts of the request message to decrypt. See the topic"Configuring the client for response decryption: decrypting the message parts" on page 1722 if you have not previously specified this information.

Configuring XML basic authentication for Version 5.x web services with an assembly tool

With the basic authentication (BasicAuth) authentication method, the request sender generates a BasicAuth security token using a callback handler. The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a Java™ Authentication and Authorization Service (JAAS) login module. Trust is established by using user name and password validation.

Securing web services for Version 5.x applications using basic authentication

With the basic authentication (BasicAuth) authentication method, the request sender generates a BasicAuth security token using a callback handler. The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. Trust is established by using user name and password validation.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. BasicAuth authentication is one of these methods. You might also secure your web services using any of the following methods:

- XML digital signature
- XML encryption
- BasicAuth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

To use BasicAuth authentication to secure web services, complete the following tasks:

Procedure

1. Secure the client for BasicAuth authentication.
   a. Configure the client for basic authentication: specifying the method
   b. Configure the client for basic authentication: collecting the authentication information
2. Secure the server for BasicAuth authentication.
   a. Configure the server to handle basic authentication
   b. Configure the server to validate basic authentication information

Results

After completing these steps, you have secured your web services using BasicAuth authentication.

Configuring the client for basic authentication: specifying the method

Basic authentication (BasicAuth) refers to the user ID and password of a valid user in the registry of the target server. BasicAuth information can be collected in many ways, including through an administrative console prompt, a standard in (Stdin) prompt, or specified in the bindings that prevents user interaction.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

For more information on BasicAuth authentication, see: “BasicAuth authentication method” on page 1727.

About this task

Attention: WebSphere Application Server supports nonce (randomly generated token) with BasicAuth authentication. For more information, see Nonce.

Complete the following steps to specify BasicAuth as the authentication method:

Procedure

1. Launch an assembly tool. See more information on the assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.

5. Click the **WS Extensions** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section. The only valid login configuration choices for a pure client are **BasicAuth** and **Signature**.
7. Select **BasicAuth** to authenticate the client using a user ID and a password. This user ID and password must be specified in the target user registry. The other choice, Signature, attempts to authenticate the client using the certificate used to digitally sign the message.

What to do next

For more information on getting started with the web services client editor within the assembly tool, see either of the following topics:

- "Configuring the client security bindings using an assembly tool" on page 1708
- Configuring the security bindings on a server acting as a client using the administrative console

After you specify the BasicAuth authentication method, you must specify how to collect the authentication information. See "Configuring the client for basic authentication: collecting the authentication information" on page 1728.

BasicAuth authentication method:

When you use the BasicAuth authentication method, the security token that is generated is a <wsse:UsernameToken> element with <wsse:Username> and <wsse:Password> elements.

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

WebSphere Application Server supports text passwords but not password digest because passwords are not stored and cannot be retrieved from the server. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

BasicAuth token generation

The request sender generates a BasicAuth security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler that is used is specified in the <LoginBinding> element of the bindings file, `ibm-webservicesclient-bnd.xmi`. The following callback handler implementations are provided with WebSphere Application Server and can be used with the BasicAuth authentication method:

- com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
- com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler

You can add your own callback handlers that implement the `javax.security.auth.callback.CallbackHandler` method.

BasicAuth token validation

The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a JAAS login module. The <wsse:Username> and <wsse:Password> elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the <LoginMapping> element of the bindings file. Default bindings are specified in the `ws-security.xml` file. However, you can override these

bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName value. The CallbackHandlerFactory option specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl CallbackHandlerFactory implementation. The ConfigName value specifies a JAAS configuration name entry. WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file for a match. WebSphere Application Server provides the WSLogin default configuration entry, which is suitable for the BasicAuth authentication method.

Configuring the client for basic authentication: collecting the authentication information

The basic authentication (BasicAuth) method refers to the user ID and the password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a user interface prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction.

About this task

Note: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

For more information on BasicAuth authentication, see “BasicAuth authentication method” on page 1727.

Complete this task to specify the authentication information needed for BasicAuth authentication:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Binding** tab, which is located at the bottom of deployment descriptor editor within the assembly tool.
6. Expand the **Security request sender binding configuration** > **Login binding** section.
7. Click **Edit** or **Enable** to view the login binding information. The login binding information displays and enter the following information:

   Authentication method
   Specifies the type of authentication. Select **BasicAuth** to use basic authentication.

   Token value type URI and Token value type local name
   When you select **BasicAuth**, you cannot edit the token value type URI and the local name values. Specifies values for custom authentication types. For BasicAuth authentication, leave these values blank.

   Callback handler
   Specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting the BasicAuth information. You can use the following default implementations for the callback handler:

   com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
   This implementation is used for non-user interface console prompts.

Restriction: This implementation prompts for the user name and password and reads them into the configuration from standard in. If you have a multi-threaded client and multiple threads attempt to read from standard in at the same time, all the threads will not successfully obtain the user name and password information. Therefore, you cannot use the `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler` implementation with a multi-threaded client where multiple threads might attempt to obtain data from standard in concurrently.

com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler
This implementation is used for user interface panel prompts.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This implementation is used when you plan to always enter the user ID and password in the BasicAuth user ID and password section that follows.

Basic Authentication user ID and Basic Authentication password
Specifies values for the BasicAuth user ID and password, regardless of the default callback handler indicated previously, these user ID and password values are used to authenticate to the server for the Web Services Security authentication. If you leave these values blank, use either the `GUIPromptCallbackHandler` or the `StdinPromptCallbackHandler` implementation, but only on a pure client. Always fill-in these values for any web service that acts as a client to another web service that you want to specify for BasicAuth for authentication downstream. If you want the client identity of the originator to flow downstream, configure the web service client to use either ID assertion or Lightweight Third Party Authentication (LTPA).

Property
Specifies properties with name and value pairs for custom callback handlers to use. For BasicAuth authentication, you do not need to enter any information. To enter a new property, click **Add** and enter the new property and value.

Results

Other basic authentication entries: There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the Web Services Security basic authentication section overrides the basic authentication information specified in the Port Qualified Name Binding Details section for authorizing the web service.

For a server that acts as a client, do not specify a user interface or non-user interface prompt callback handler. To configure BasicAuth authentication from one web service to a downstream web service, select the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHander` implementation and explicitly specify the BasicAuth user ID and password. If you want the client identity of the originator to flow downstream, configure the web service client to use ID assertion.

What to do next

To use the BasicAuth authentication method, you must specify the method in the Login configuration section of the assembly tool . See “Configuring the client for basic authentication: specifying the method” on page 1726 if you have not previously specified this information.

Identity assertion authentication method:

When using the identity assertion (IDAssertion) authentication method, the security token generated is a <wsse:UsernameToken> element that contains a <wsse:Username> element.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* applications.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. These two operations, token generation and token validation operations, are described in the following sections.

Identity assertion token validation:

The request receiver retrieves the IDAssertion security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. With identity assertion, special processing is required to establish trust before asserting the identity as the established identity of the running thread. This special processing is defined by the <IDAssertion> element in the deployment descriptor file, `ibm-webservices-ext.xmi`. If all the validation checks are successful, the asserted identity is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the<LoginMapping> element of the bindings file. Default bindings are specified in the `ws-security.xml` file. However, you can override these bindings using the application specific `ibm-webservices-bnd.xmi` file. The configuration information consists of CallbackHandlerFactory and a ConfigName. CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` CallbackHandlerFactory implementation. ConfigName specifies a JAAS configuration name entry.

WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found it searches the `wsjaas.conf` file. WebSphere Application Server provides the system.wssecurity.IDAssertion default configuration entry, which is suitable for the identity assertion authentication method.

The <IDAssertion> element in the `ibm-webservices-ext.xmi` deployment descriptor file specifies the special processing required when using the identity assertion authentication method. The <IDAssertion> element is composed of two sub-elements: <IDType> and <TrustMode>.

The <IDType> element specifies the method for asserting the identity. The supported values for asserting the identity are:

- Username
- Distinguished name (DN)
- X.509 certificate

When <IDType> is *username*, a username token (for example, Bob) is provided. This user name is mapped to a user in the user registry and is the asserted identity after successful trust validation. When the <IDType> value is *DN*, a user name token containing a distinguished name is provided (for example, `cn=Bob Smith, o=ibm, c=us`). This DN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation. When the <IDType> is *X509Certificate*, a binary security token containing an X509 certificate is provided and the SubjectDN value from the certificate (for example, `cn=Bob Smith, o=ibm, c=us`) is extracted. This SubjectDN value is mapped to a user in the user registry and this user is the asserted identity after successful trust validation.

The `<TrustMode>` element specifies how the trust authority, or asserting authority, provides trust information. The supported values are:

- Signature
- BasicAuth

- No value specified

When the <TrustMode> value is `Signature`, the signature is validated. Then, the signer (for example, `cn=IBM Authority, o=ibm, c=us`) is mapped to an identity in the user registry (for example, IBMAuthority). To ensure that the asserting authority is trusted, the mapped identity (for example, IBMAuthority) is validated against a list of trusted identities. When the <TrustMode> element is `BasicAuth`, there is a user name token with a user name and password, which is the user name and password of the asserting authority.

The user name and password are validated. If they are successfully validated, that user name (for example, IBMAuthority) is validated against a list of trusted identities. If a value is not specified for `<TrustMode>`, trust is presumed and additional trust validation is not performed. This type of identity assertion is called *presumed trust mode*. Use the presumed trust mode only in an environment where the trust is established using some other mechanism.

If all the validations described previously succeed, the asserted identity (for example, Bob) is set as the identity of the running thread. If any of the validations fail, the request is rejected with a SOAP fault exception.

Configuring the server to handle basic authentication information

Basic authentication (BasicAuth) refers to the user ID and the password of a valid user in the registry of the target server. After a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

About this task

Complete the following steps to configure the server to handle BasicAuth authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the web services editor within an assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. You can select the following options:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **Lightweight Third Party Authentication (LTPA)**
7. Select **BasicAuth** to authenticate the client with a user ID and a password. The client must specify a valid user ID and password in the server user registry. If the user ID and the password supplied are not valid, an exception is provided, and the request ends without invoking the resource.

You can select multiple login configurations, which means that different types of security information might be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. Problems can occur if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token. For ID assertion to work properly, list ID assertion ahead of BasicAuth in the processing list or the BasicAuth processing overrides the IDAssertion processing.

What to do next

After you specify how the server handles BasicAuth authentication information, you must specify how the server validates the authentication information. See the task for configuring the server to validate BasicAuth authentication if you have not previously specified this information.

Configuring the server to validate basic authentication information

Basic authentication (BasicAuth) refers to the user ID and the password of a valid user in the registry of the target server. You can specify how the server validates the BasicAuth information.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

After a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and the password supplied is invalid, an exception is thrown and the request ends without invoking the resource.

About this task

Complete the following steps to specify how the server validates the BasicAuth authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab, which is located at the bottom of the web services editor within an assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog is displayed. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select **BasicAuth** to use basic authentication.

   Configuration name
   Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the BasicAuth authentication method, enter `WSLogin` for the JAAS login Configuration name.

   Use token valid type
   Determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.

Token value type URI and Token value type URI local name
When you select BasicAuth, you cannot edit the token value type URI and local name values. Specifies custom authentication types. For BasicAuth authentication leave these fields blank.

Callback handler factory class name
Creates a JAAS CallbackHandler implementation that understands the following callbacks:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

Callback handler factory property name and Callback handler factory property value
Specifies callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For BasicAuth, you do not need to enter any property values.

Login mapping property name and Login mapping property value
Specifies properties for a custom login mapping. For the default implementations including BasicAuth, leave these fields blank.

What to do next

You must specify how the server handles the BasicAuth authentication method. See the task for configuring the server to handle basic authentication if you have not previously specified this information.

Configuring identity assertion for Version 5.x web services with an assembly tool

With the identity assertion authentication method, the security token generates a <wsse:UsernameToken> element that contains a <wsse:Username> element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. Trust is established through the use of a security token rather than through user name and password validation.

Securing web services for Version 5.x applications using identity assertion authentication

With the identity assertion authentication method, the security token generates a <wsse:UsernameToken> element that contains a <wsse:Username> element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. Unlike BasicAuth authentication, trust is established through the use of a security token rather than through user name and password validation.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. Identity assertion authentication is one of these methods. You might also secure your web services using any of the following methods:

- XML digital signature
- XML encryption
- BasicAuth authentication

- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

To use identity assertion authentication to secure web services, complete the following tasks:

Procedure

1. Secure the client for identity assertion authentication.
   a. “Configuring the client for identity assertion: specifying the method”
   b. “Configuring the client for identity assertion: collecting the authentication method” on page 1735
2. Secure the server for identity assertion authentication.
   a. “Configuring the server to handle identity assertion authentication” on page 1736
   b. “Configuring the server to validate identity assertion authentication information” on page 1738

Results

After completing these steps, you have secured your web services by using identity assertion authentication.

Configuring the client for identity assertion: specifying the method

You can configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a web service acting as a client to a downstream web service.

In order for the downstream web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. See topics about trusted ID evaluators.

Complete the following steps to specify identity assertion as the authentication method:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extension** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section.

7. Select **IDAssertion** as the authentication method. For more conceptual information on identity assertion authentication, see Identity assertion in a SOAP message.
8. Expand the **IDAssertion** section.
9. For the ID type, select **Username**. This value works with all registry types and originating authentication methods.
10. For the trust mode, select either **BasicAuth** or **Signature**.
    - By selecting **BasicAuth**, you must include basic authentication information (user ID and password), which the downstream web service has specified in the trusted ID evaluator as a trusted user ID. See “Configuring the client for signature authentication: collecting the authentication information” on page 1742 to specify the user ID and password information.
    - By selecting **Signature** the certificate configured in the signature information section used to sign the data also is that is used as the trusted subject. The Signature is used to create a credential and user ID, which the certificate mapped to the downstream registry, is used in the trusted ID evaluator as a trusted user ID.

What to do next

See “Configuring the client security bindings using an assembly tool” on page 1708 for more information on the web services client editor within the assembly tool.

After you specify identity assertion as the authentication method used by the client, you must specify how to collect the authentication information. See “Configuring the client for identity assertion: collecting the authentication method” for more information.

Configuring the client for identity assertion: collecting the authentication method

You can configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a web service acting as a client to a downstream web service.

In order for the downstream web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. See the information on trusted ID evaluators.

Complete the following steps to specify how the client collects the authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.

5. Click the **WS Binding** tab, which is located at the bottom of the Deployment Descriptor Editor within an assembly tool.
6. Expand the **Security request sender binding configuration** > **Login binding** section.
7. Click **Edit** to view the login binding information and select **IDAssertion**. The login binding dialog is displayed. Select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use identity assertion.

   Token value type URI and Token value type Local name
   When you select IDAssertion, you cannot edit the token value type Universal Resource Identifier (URI) and the local name. Specifies custom authentication types. For IDAssertion authentication, leave these values blank.

   Callback handler
   Specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the BasicAuth information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler` implementation for IDAssertion.

   Basic authentication User ID and Basic authentication Password
   In this field, the trust mode entered in the extensions is BasicAuth. Specifies the trusted user ID and password in these fields. The user ID specified must be an ID that is trusted by the downstream web service. The web service trusts the user ID if it is entered as a trusted ID in a trusted ID evaluator in the downstream web service bindings. If the trust mode entered in the extensions is `Signature`, you do not need to specify any information in this field.

   Property name and Property value
   Specifies properties with name and value pairs, for use by custom callback handlers. For IDAssertion, you do not need to specify any information in this field.

What to do next

To use the identity assertion authentication method, you must specify the method in the Security extensions section of an assembly tool. See “Configuring the client for identity assertion: specifying the method” on page 1734 if you have not previously specified this information.

Configuring the server to handle identity assertion authentication

The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream web service. You can configure identity assertion authentication for the server. Do not attempt to configure identity assertion from a pure client.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

For the downstream web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluator. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns `true` or `false` that this ID is trusted. After it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to configure the server to handle identity assertion authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. The options you can select are:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **LTPA** (Lightweight Third Party Authentication)
7. Select **IDAssertion** to authenticate the client using the identity assertion data provided.

   The user ID of the client must be in the target user registry or repository, which is configured on the **Security** > **Global security** panel in the administrative console for WebSphere Application Server. You can select multiple login configurations, which means that different types of security information can be received at the server. The order in which the login configurations are added determines the processing order when a request is received. Problems can occur if you have multiple login configurations added that have common security tokens. For example, ID assertion contains a BasicAuth token, which is the trusted token. For ID assertion to work properly, you must list ID assertion ahead of BasicAuth in the list or BasicAuth processing overrides ID assertion processing.
8. Expand the **IDAssertion** section and select both the **ID Type** and the **Trust Mode**.
   a. For ID Type, the options are:
      - **Username**
      - **DN** (distinguished name)
      - **X509certificate**

      These choices are just preferences and are not guaranteed. Most of the time the Username option is used. You must choose the same ID Type as the client.
   b. For Trust Mode, the options are:
      - **BasicAuth**
      - **Signature**

      The Trust Mode refers to the information sent by the client as the trusted ID.
      1) If you select **BasicAuth**, the client sends basic authentication data (user ID and password). This BasicAuth data is authenticated to the configured user registry. When the authentication occurs successfully, the user ID must be part of the trusted ID evaluator trust list.
      2) If you select **Signature**, the client signing certificate is sent. This certificate must be mappable to the configured user registry. For **Local OS**, the common name (CN) of the distinguished name (DN) is mapped to a user ID in the registry. For **Lightweight Directory Access Protocol (LDAP)**, the DN is mapped to the registry for the ExactDN mode. If it is in the CertificateFilter mode, attributes are mapped accordingly. In addition, the user name from the credential generated must be in the Trusted ID Evaluator trust list.

What to do next

For more information on getting started with the Web Services Editor within an assembly tool, see “Configuring the server security bindings using an assembly tool” on page 1711.

After you specify how the server handles identity assertion authentication information, you must specify how the server validates the authentication information. See the task for configuring the server to validate identity assertion authentication information.

Configuring the server to validate identity assertion authentication information

The purpose of identity assertion is to assert the authenticated identity of the originating client from a web service to a downstream Web service.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

Use this task to configure identity assertion authentication. Do not attempt to configure identity assertion from a pure client.

For the downstream web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream web service configuration. For more information on trusted ID evaluators, see the topic about the trusted ID evaluator. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns a `true` or `false` response that this ID is trusted. After it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to validate the identity assertion authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective > J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Binding Configurations tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information. Click **Add** to add new login mapping information. The login mapping dialog is displayed. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select **IDAssertion** to use basic authentication.

   Configuration name
   Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the IDAssertion authentication method, enter `system.wssecurity.IDAssertion` for the Java Authentication and Authorization Service (JAAS) login configuration name.

Use token value type
Determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.

Token value type URI and Token value type local name
When you select ID assertion, you cannot edit the token value type URI and local name values. Specifies custom authentication types. For the ID assertion authentication method, leave these values blank.

Callback handler factory class name
Ccreates a JAAS CallbackHandler implementation that understands the following callbacks:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

For any of the default authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including IDAssertion:

```
com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
```

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property name and Callback handler factory property value
Specifies callback handler properties for custom callback handler factory implementations. The default callback handler factory implementation does not need any specified properties. For ID assertion, leave these values blank.

Login mapping property name and Login mapping property value
Specifies properties for a custom login mapping. For the default implementations including IDAssertion, leave these values blank.

8. Expand the **Trusted ID evaluator** section.
9. Click **Edit** to see a dialog that displays all the trusted ID evaluator information. The following table describes the purpose of this information.

Class name
Refers to the implementation of the trusted ID evaluator that you want to use. Enter the default implementation as

```
com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl
```

If you want to implement your own trusted ID evaluator, you must implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

Property name
Represents the name of this configuration. Enter `BasicIDEvaluator`.

Property value
Defines the name and value pairs that can be used by the trusted ID evaluator implementation. For the default implementation, the trusted list is defined here. When a request comes in and the trusted ID is verified, the user ID, as it appears in the user registry, must be listed in this property. Specify the property as a name and value pair where the name is *trustedId_n*. *n* is an integer starting from 0 and the value is the user ID associated with that name. An example list with the trusted names include two properties.

For example: `trustedId_0 = user1, trustedId_1 = user2`. The previous example means that both `user1` and `user2` are trusted. `user1` and `user2` must be listed in the configured user registry

10. Expand the **Trusted ID evaluator reference** section.

11. Click **Enable** to add a new entry. The text you enter for the **Trusted ID evaluator reference** must be the same as the name entered previously in the **Trusted ID evaluator**. Make sure that the name matches exactly because the information is case sensitive. If an entry is already specified, you can change it by clicking **Edit**.

What to do next

You must specify how the server handles the identity assertion authentication method. See “Configuring the server to handle identity assertion authentication” on page 1736 if you have not previously specified this information.

Configuring signature authentication for Version 5.x web services with an assembly tool

With the signature authentication method, the request sender generates a signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The request receiver retrieves the Signature security token from the SOAP message and validates it using a Java™ Authentication and Authorization Service (JAAS) login module.

Securing web services for version 5.x applications using signature authentication

WebSphere Application Server provides several different methods to secure your web services. XML digital signature is one of these methods.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

You can secure your web services by using any of the following methods:

- XML digital signature
- XML encryption
- BasicAuth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

About this task

With the signature authentication method, the request sender generates a signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The request receiver retrieves the Signature security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. To use signature authentication to secure Web services, complete the following tasks:

Procedure

1. Secure the client for signature authentication.
   a. “Configuring the client for signature authentication: specifying the method” on page 1741.
   b. “Configuring the client for signature authentication: collecting the authentication information” on page 1742.
2. Secure the server for signature authentication.
   a. “Configuring the server to support signature authentication” on page 1744.

b. “Configuring the server to validate signature authentication information” on page 1745.

Results

After completing these steps, you have secured your web services using signature authentication.

Configuring the client for signature authentication: specifying the method

Signature authentication, the use of an X.509 certificate to login on the target server, can be configured.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server. For more information on signature authentication, see “Signature authentication method.”

Complete the following steps to specify signature as the authentication method:

Procedure

1. Launch an assembly tool. For more information, read about assembly tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF** .
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extension** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section. The following login configuration options are valid for a managed client and Web services acting as a client are:

   BasicAuth
   Use this option for a managed client.

   Signature
   Use this option for a managed client.

   IDAssertion
   Use this option for web services acting as a client.
7. Select **Signature** to authenticate the client using the certificate used to digitally sign the request.

Results

For more information on getting started with the web services client editor within the assembly tool, see “Configuring the client security bindings using an assembly tool” on page 1708.

After you specify signature as the authentication method, you must specify how to collect the authentication information. See “Configuring the client for signature authentication: collecting the authentication information” on page 1742 for more information.

Signature authentication method:

Signature authentication refers to an X.509 certificate that is sent by the client to the server. The certificate is used to authenticate to the user registry that is configured at the server. When using the signature authentication method, the security token is generated with a ds:Signature and a wsse:BinarySecurityToken element.

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

Signature token generation

The request sender generates a Signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler is specified in the <LoginBinding> element of the bindings file, `ibm-webservicesclient-bnd.xmi`. WebSphere Application Server provides the following callback handler implementation that can be used with the Signature authentication method:
com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler

You can add your own callback handlers that implement the javax.security.auth.callback.CallbackHandler implementation.

Security token validation

The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. The <ds:Signature> and <wsse:BinarySecurityToken> elements in the security token are used to perform the validation. If the validation is successful, the login module returns a Java Authentication and Authorization Service (JAAS) Subject. This Subject then is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the `<LoginMapping>` element of the bindings file. Default bindings are specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImp CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches in the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file. WebSphere Application Server provides the system.wssecurity.Signature default configuration entry, which is suitable for the signature authentication method.

Configuring the client for signature authentication: collecting the authentication information

Signature authentication refers to an X.509 certificate that is sent by the client to the server. The certificate is used to authenticate to the user registry that is configured at the server. The client collects the authentication information for signature authentication.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

You can configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server.

Complete the following steps to specify how the client collects the authentication information for signature authentication:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the WS Binding tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Security request sender binding configuration** > **Signing information** and click **Edit** to modify the signing key name and signing key locator. To create new signing information, click **Enable**. The certificate that is sent to log in at the server is the one configured in the Signing Information section. Review the key locator information to understand how the signing key name maps to a key within the key locator entry.

   The following list describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following web address: http://www.w3.org/TR/xmldsig-core

   Canonicalization method algorithm
   Canonicalizes the <SignedInfo> element before it is digested as part of the signature operation.

   Digest mehod algorithm
   Represents the algorithm that is applied to the data after transforms are applied, if specified, to yield the <DigestValue> element. The signing of the <DigestValue> element binds the resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

   Signature method algorithm
   Represents the algorithm that is used to convert the canonicalized <SignedInfo> element value into the <SignatureValue> value. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

   Signing key name
   Represents the key entry that is associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.

   Signing key locator
   Represents a reference to a key locator implementation.
7. Expand the **Security request sender binding configuration** > **Login binding** section.
8. Click **Edit** to view the login binding information. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select **Signature** to use signature authentication.

Token value type URI and Token value type URI local name
When you select **Signature**, you cannot edit token value type Uniform Resource Identifier (URI) and local name values. Specifies custom authentication types. For signature authentication, leave these fields blank.

Callback handler
Specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting signature information. Enter the following callback handler for signature authentication:
`com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

This callback handler is used because the signature method does not require user interaction.

Basic authentication user ID and Basic authentication password
Leave the BasicAuth fields blank when signature authentication is used.

Property name and property value
This field enables you to enter properties and name and value pairs for use by custom callback handlers. For signature authentication, do not enter any information.

What to do next

Other customization entries: There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the Web Services Security signature authentication section overrides the basic authentication information specified in the Port Qualified Name Binding Details section for authorizing the Web service.

To use the signature authentication method, you must specify the authentication method in the Login configuration section of an assembly tool.

Configuring the server to support signature authentication

Signature authentication refers to an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. After a request is received by the server that contains the certificate, the server needs to log in to form a credential. The credential is used for authorization. You can configure signature authentication at the server.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

If the certificate supplied cannot be mapped to an entry in the user registry, an exception is provided and the request ends without invoking the resource.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective by clicking **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Extensions** tab, which is located at the bottom of the Web Services Editor within the assembly tool.

6. Expand the **Request receiver service configuration details** > **Login configuration** section. You can select from the following options:
   - BasicAuth
   - Signature
   - ID assertion
   - Lightweight Third Party Authentication (LTPA)
7. Select **Signature** to authenticate the client using an X509 certificate. The certificate that is sent from the client is the certificate that issued for signing the message. You must be able to map this certificate to the configured user registry. For Local operating system (OS) registries, the common name (cn) of the distinguished name (DN) is mapped to a user ID in the registry. For Lightweight Directory Access Protocol (LDAP), you can configure multiple mapping modes:
   - `EXACT_DN` is the default mode that directly maps the DN of the certificate to an entry in the LDAP server.
   - `CERTIFICATE_FILTER` is the mode that provides the LDAP advanced configuration with a place to specify a filter that maps specific attributes of the certificate to specific attributes of the LDAP server.

What to do next

For more information on getting started with the web services editor within the assembly tool, see "Configuring the server security bindings using an assembly tool" on page 1711.

After you specify how the server handles signature authentication information, you must specify how the server validates the authentication information. See the task for configuring the server to validate signature authentication.

Configuring the server to validate signature authentication information

Signature authentication refers to an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. After a request is received by the server that contains the certificate, the server needs to log in to form a credential. The credential is used for authorization. You can validate signature authentication at the server.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource.

Complete the following steps to configure the server to validate signature authentication:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective by clicking **Window** > **Open perspective** > **Other** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the **Binding Configurations** tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.

7. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog is displayed and you select (or enter) the following information:

   Authentication method
   Specifies the type of authentication. Select **Signature** to use signature authentication.

   Configuration name
   Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the signature authentication method, enter `system.wssecurity.Signature` for the JAAS login configuration name. This specification logs in with the `com.ibm.wsspi.wssecurity.auth.module.SignatureLoginModule` JAAS login module.

   Use token value type
   Determines if you want to specify a custom token type. For the default authentication method selections, you can leave this field blank.

   URI and local name
   When you select Signature method, you cannot edit the token value type URI and local name values. Specifies custom authentication types. For signature authentication, you can leave this field blank.

   Callback handler factory class name
   Creates a JAAS CallbackHandler implementation that understands the following callback handlers:

   - javax.security.auth.callback.NameCallback
   - javax.security.auth.callback.PasswordCallback
   - com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
   - com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
   - com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

   For any of the default authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including signature:

```
com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
```

   This implementation creates the correct callback handler for the default implementations.

   Callback handler factory property name and callback handler factory property value
   Specifies callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For signature, you can leave this field blank.

   Login mapping property name and login mapping property value
   Specifies properties for a custom login mapping to use. For the default implementations including signature, you can leave this field blank.

What to do next

Specify how the server handles the signature authentication method. See “Configuring the server to support signature authentication” on page 1744 if you have not previously specified this information.

Configuring pluggable tokens for Version 5.x web services with an assembly tool

WebSphere Application Server provides several different methods to secure your web services, including the pluggable token method. To use pluggable tokens to secure your web services, you must configure both the client request sender and the server request receiver.

Securing web services for version 5.x applications using a pluggable token

To use pluggable tokens to secure your web services, you must configure both the client request sender and the server request receiver. You can configure your pluggable tokens using the WebSphere Application Server administrative console.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

WebSphere Application Server provides several different methods to secure your web services. A pluggable token is one of these methods. You might secure your Web services by using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

Complete the following steps to secure your web services using a pluggable token:

Procedure

1. Generate a security token using the Java Authentication and Authorization Service (JAAS) CallbackHandler interface. The Web Services Security runtime uses the JAAS CallbackHandler interface as a plug-in to generate security tokens on the client side or when web services are acting as a client.
2. Configure your pluggable token. For more information, see the following tasks:
   - "Configuring pluggable tokens using an assembly tool"
   - Configuring pluggable tokens using the administrative console

Configuring pluggable tokens using an assembly tool

The following information describes how to configure a pluggable token using an assembly tool.

Before you begin

Important: There is an important distinction between Version 5.*x* and Version 6 and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

The pluggable token is required for the request sender and request receiver because they are a pair. The request sender and the request receiver must match for the receiver to accept a request.

Prior to completing these steps, it is assumed that you have already created a web service that is based on the Java Platform, Enterprise Edition (Java EE) specification. See either of the following topics for an introduction of how to manage Web Services Security binding information for the server:

- “Configuring the server security bindings using an assembly tool” on page 1711
- Configuring the server security bindings using the administrative console

About this task

You must specify the security constraints in the `ibm-webservicesclient-ext.xmi` and the `ibm-webservices-ext.xmi` files for the required tokens using an IBM assembly tool.

Complete the following steps to configure the request sender using the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files:

Procedure

1. Launch an assembly tool. For more information, read about assembly tools.
2. Switch to the Java EE perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Extension** tab. The web service client security extensions editor is displayed.
   a. Under Service References, select an existing service reference or click **Add** to create a new reference.
   b. Under Port Qname Bindings, select an existing port qualified name for the selected service reference or click **Add** to create a new port name binding.
   c. Under Request Sender Configuration: Login Configuration, select an exiting authentication method or type in a new one in the editable list box (Lightweight Third Party Authorization (LTPA) is a supported token generation when web services are acting as client).
   d. Click **File** > **Save** to save the changes.
6. Click the **Web services client binding** tab. The web services client binding editor is displayed.
   a. Under Port qualified name binding, select an existing entry or click **Add** to add a new port name binding. The web services client binding editor displays for the selected port.
   b. Under Login binding, click **Edit** or **Enable**. The Login Binding dialog box is displayed.
      1) In the Authentication Method field, enter the authentication method. The authentication method that you enter in this field must match the authentication method defined on the **Security Extension** tab for the same web service port. This field is mandatory.
      2) (Optional) Enter the token value type information in the URI and Local name fields. These fields are ignored for the BasicAuth, Signature, and IDAssertion authentication methods, but required for other authentication methods. The token value type information is inserted into the <wsse:BinarySecurityToken>@ValueType element for binary security token and is used as the namespace for the XML-based token.
      3) Enter an implementation of the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface. This field is mandatory.
      4) Enter the basic authentication information in the **User ID** and **Password** fields. The basic authentication information is passed to the construct of the `CallbackHandler` implementation. The use of the basic authentication information depends on the implementation of `CallbackHandler`.
      5) In the Property field, add name and value pairs. These pairs are passed to the construct of the `CallbackHandler` implementation as `java.util.Map` values.
      6) Click **OK**.

      Click **Disable** under Login binding on the **Web services client port binding** tab to remove the authentication method login binding.
   c. Click **File** > **Save** to save the changes.

7. In the Package Explorer window, right-click the `webservices.xml` file and click **Open with** > **Web services editor**. The Web Services window displays.
   a. Click the **Security extensions** tab. The Web Service Security extensions editor is displayed.
      1) Under Web Services Description Extension, select an existing service reference or click **Add** to create a new extension.
      2) Under Port Component Binding, select an existing port qualified name for the selected service reference or click **Add** to create a new one.
      3) Under Request Receiver Service Configuration Details: Login Configuration, select an exiting authentication method or click **Add** and enter a new method in the **Add AuthMethod** field that displays. You can select multiple authentication methods for the request receiver. The security token of the incoming message is authenticated against the authentication methods in the order that they are specified in the list. Click **Remove** to remove the selected authentication method or methods.
   b. Click **File** > **Save** to save the changes.
   c. Click the **Bindings** tab. The web services bindings editor is displayed.
      1) Under web service description bindings, select an existing entry or click **Add** to add a new web services descriptor.
      2) Click the **Binding configurations** tab. The web services binding configurations editor is displayed for the selected web services descriptor.
      3) Under Request receiver binding configuration details: login mapping, click **Add** to create a new login mapping or click **Edit** to edit the selected login mapping. The Login mapping dialog is displayed.
         a) In the Authentication method field, enter the authentication method. The information entered in this field must match the authentication method defined on the **Security Extensions** tab for the same web service port. This field is mandatory.
         b) In the **Configuration name** field, enter a JAAS login configuration name. This field is mandatory. You must define the JAAS login configuration name in the WebSphere Application Server administrative console under **Security** > **Global security**. Under Authentication, click **Java Authentication and Authorization Service** > **Application logins**. For more information, read about configuring programmatic logins for Java Authentication and Authorization Service.
         c) (Optional) Select **Use Token value type** and enter the token value type information in the URI and Local name fields. This information is optional for BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is used to validate the <wsse:BinarySecurityToken>@ValueType element for binary security tokens and to validate the namespace of the XML-based token.
         d) Under Callback Handler Factory, enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface in the Class name field. This field is mandatory.
         e) Under Callback Handler Factory property, click **Add** and enter the name and value pairs for the Callback Handler Factory Property. These name and value pairs are passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init()` method. The use of these name and value pairs is determined by the `CallbackHandlerFactory` implementation.
         f) Under Login Mapping Property, click **Add** and enter the name and value pairs for the Login mapping property. These name and value pairs are available to the JAAS Login Modules through the `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback interface. Click **Remove** to delete the selected login mapping.
         g) Click **OK**.
   d. Click **File** > **Save** to save the changes.

Results

The previous steps define how to configure the request sender to create security tokens in the SOAP message and to configure the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

What to do next

After you configure pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:

- Configuring the client for LTPA token authentication: Specifying LTPA token authentication
- Configuring the client for LTPA token authentication: Collecting the authentication information
- Configuring the server to handle LTPA token authentication
- Configuring the server to validate LTPA token authentication information

Configuring the client for LTPA token authentication: specifying LTPA token authentication

To configure Lightweight Third-Party Authentication (LTPA) token authentication, specify LTPA token authentication. Only configure the client for LTPA token authentication if the authentication mechanism configured in WebSphere Application Server is LTPA.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Only configure the client for LTPA token authentication if the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a web service calls a downstream web service, you can configure the first web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a web service acting as a client to a downstream web service. For the downstream web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify LTPA token as the authentication method:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **Extensions** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Request sender configuration** > **Login configuration** section.
7. Select **LTPA** as the authentication method. For more conceptual information on LTPA authentication, see “Lightweight Third Party Authentication” on page 1754.

What to do next

After you specify LTPA token as the authentication method, you must specify how to collect the LTPA token information. See “Configuring the client for LTPA token authentication: collecting the authentication method information” for more information.

Configuring the client for LTPA token authentication: collecting the authentication method information

To configure Lightweight Third-Party Authentication (LTPA) token authentication, collect the LTPA token authentication information. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a web service calls a downstream web service, you can configure the first web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a web service acting as a client to a downstream web service. In order for the downstream web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify how to collect the LTPA token authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **Application Client Projects** > ***application_name*** > **appClientModule** > **META-INF**.
4. Right-click the `application-client.xml` file, select **Open with** > **Deployment descriptor editor**.
5. Click the **WS Bindings** tab, which is located at the bottom of the deployment descriptor editor within the assembly tool.
6. Expand the **Security request sender binding configuration** > **Login binding** section.
7. Click **Edit** to view the login binding information and select **LTPA**. If LTPA is not already there, enter it as an option. The login binding dialog is displayed. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select **LTPA** to use identity assertion.

   Token value type URI and token value type local name
   When you select **LTPA**, you must edit the token value type **URI** (Uniform Resource Identifier) and the **local name** fields. Specifies values for custom authentication types, which are authentication methods not mentioned in the specification. For the token value type **URI** field, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For the **local name** field, enter the following string: `LTPA`.

   Callback handler
   Specifies the Java Authentication and Authorization Service (JAAS) callback handler

implementation for collecting the LTPA information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler` implementation for LTPA.

Basic authentication user ID and basic authentication password
For LTPA, you can leave these fields empty. However, when you omit this information, the LTPA `CallbackHandler` implementation attempts to obtain the LTPA token from the invocation (RunAs) credential. If an invocation (RunAs) credential does not exist, then the LTPA token is not propagated.

Property name and property value
For LTPA, you can leave these fields blank.

What to do next

See "Configuring the client for LTPA token authentication: specifying LTPA token authentication" on page 1750 if you have not previously specified this information.

Configuring the server to handle LTPA token authentication information

Lightweight Third-Party Authentication (LTPA) is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first web service, which authenticated the originating client, to the downstream web service. You can configure the server for LTPA token authentication.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure LTPA. Do not attempt to configure LTPA from a pure client. After the downstream web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys that are used by both the sending and receiving servers must be the same.

Complete the following steps to specify that LTPA is the authentication method. The authentication method indicated in these steps must match the authentication method that is specified for the client.

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java Platform, Enterprise Edition (Java EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Extensions tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver service configuration details** > **Login configuration** section. You can select from the following options:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **LTPA**
7. Select **LTPA** to authenticate the client using the LTPA token received from the request.

What to do next

After you specify the authentication method, you must specify the information that the server must validate. See “Configuring the server to validate LTPA token authentication information” for more information.

Configuring the server to validate LTPA token authentication information

Lightweight Third-Party Authentication (LTPA) is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first web service, which authenticated the originating client, to the downstream web service. You can configure the server to validate LTPA token authentication.

About this task

Important: There is an important distinction between Version 5.*x* and Version 6.0.*x* and later applications. The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6.0.*x* and later applications.

This task is used to configure LTPA. Do not attempt to configure LTPA from a pure client. After the downstream web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys used by both the sending and receiving servers must be the same.

Complete the following steps to specify how the server must validate the LTPA token authentication information:

Procedure

1. Launch an assembly tool. For more information, see the related information on Assembly Tools.
2. Switch to the Java 2 Platform, Enterprise Edition (J2EE) perspective. Click **Window** > **Open Perspective** > **J2EE**.
3. Click **EJB Projects** > ***application_name*** > **ejbModule** > **META-INF**.
4. Right-click the `webservices.xml` file, and click **Open with** > **Web services editor**.
5. Click the Binding Configurations tab, which is located at the bottom of the web services editor within the assembly tool.
6. Expand the **Request receiver binding configuration details** > **Login mapping** section.
7. Click **Edit** to view the login mapping information. The login mapping information is displayed. Select or enter the following information:

   Authentication method
   Specifies the type of authentication that occurs. Select `LTPA` to use LTPA token authentication.

   Configuration name
   Specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the LTPA authentication method, enter `WSLogin` for the JAAS login configuration name. This configuration understands how to validate an LTPA token.

   Use token value type
   Determines if you want to specify a custom token type. For LTPA authentication, you must select this option because LTPA is considered a custom type. LTPA is not in the Web Services Security Specification.

   Token value type URI and local name
   Specifies custom authentication types. If you select **Use Token value type** you must enter data into the Token value Type URI (Uniform Resource Identifier) and local name fields. For

the token value type URI field, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For the local name, enter the following string: `LTPA`

Callback handler factory class name
Creates a JAAS CallbackHandler implementation that understands the following callback handlers:

- javax.security.auth.callback.NameCallback
- javax.security.auth.callback.PasswordCallback
- com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
- com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback
- com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback

For any of the default authentication methods (BasicAuth, IDAssertion, Signature, and LTPA), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including LTPA:

```
com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl
```

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property
Specifies callback handler properties for custom callback handler factory implementations. Default callback handler factory implementation does not any property specifications. For LTPA, leave this field blank.

Login mapping property
Specifies properties for a custom login mapping. For default implementations including LTPA, leave this field blank.

What to do next

See the task for configuring the server to handle LTPA token authentication information if you have not previously specified this information.

Lightweight Third Party Authentication:

When you use the lightweight third party authentication (LTPA) method, the <wsse:BinarySecurityToken> security token is generated. On the request sender side, the security token is generated by invoking a callback handler. On the request receiver side, the security token is validated by a Java Authentication and Authorization Service (JAAS) login module.

Important: The information in this article supports Version 5.*x* applications only that are used with WebSphere Application Server Version 6.0.*x* and later. The information does not apply to Version 6 and later applications.

The following information describes token generation and token validation operations.

LTPA token generation
The request sender uses a callback handler to generate an LTPA security token. The callback handler returns a security token that is inserted in the SOAP message. Specify the appropriate callback handler in the <LoginBinding> element of the bindings file (`ibm-webservicesclient-bnd.xmi`). The following callback handler implementation can be used with the LTPA authentication method:

- `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`

You can add your own callback handlers that implement the `javax.security.auth.callback.CallbackHandler` property.

When using the LTPA authentication method (or any authentication method other than BasicAuth, Signature or IDAssertion), the TokenValueType attribute of the <LoginBinding> element in the bindings file (`ibm-webservicesclient-bnd.xmi`) must be specified. The values to use for the LTPA TokenValueType attribute are:

- `uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"`
- `localName="LTPA"`

LTPA token validation

The request receiver retrieves the LTPA security token from the SOAP message and validates the message using a JAAS login module. The <wsse:BinarySecurityToken> security token is used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. Subsequently, this Subject is set as the identity of the running thread. If the validation fails, the request is rejected with a SOAP fault.

The appropriate JAAS login configuration to use is specified in the bindings file <LoginMapping> element. Default bindings specified in the `ws-security.xml` file, but these can be overridden using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory, a ConfigName and a TokenValueType attribute. The CallbackHandlerFactory specifies the name of a class to use to create the JAAS CallbackHandler object. A CallbackHandlerFactory implementation is provided (com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl). The ConfigName attribute specifies a JAAS configuration name entry. The Web Services Security run time first searches the `security.xml` file for a matching entry and if a matching entry is not found, the run time searches the `wsjaas.conf` file. A default configuration entry suitable for the LTPA authentication method is provided (`WSLogin`). An appropriate TokenValueType element is located in the LTPA LoginMapping section of the default `ws-security.xml` file.

Chapter 35. Developing web services - Transaction support (WS-Transaction)

WS-Transaction is an interoperability standard that includes the WS-AtomicTransaction, WS-BusinessActivity, and WS-Coordination specifications. The Web Services Atomic Transaction (WS-AT) support in the application server provides transactional quality of service to the web services environment. Distributed web services applications, and the resources they use, can take part in distributed global transactions. With Web Services Business Activity (WS-BA) support in the application server, web services on different systems can coordinate activities that are more loosely coupled than atomic transactions. Such activities can be difficult or impossible to roll back atomically, and therefore require a compensation process if an error occurs. Web Services Coordination (WS-COOR) specifies a CoordinationContext and a Registration service with which participant web services can enlist to take part in the protocols that are offered by specific coordination types.

Creating an application that uses the Web Services Business Activity support

To create an application component that uses the business activity support, you must set **Run EJB methods under a Business Activity scope** in the deployment descriptor of the relevant application component, and if required, create and specify a compensation handler for the application to use if there is an error. You then build the component into the application and deploy the application onto a server that has the business activity support enabled. The application component can be either an enterprise bean or a web service that is implemented as an enterprise bean.

Before you begin

For information about editing deployment descriptors by using Rational Application Developer, refer to the Rational Application Developer information.

About this task

Complete this task for an application that runs on a business-activity-enabled sever to use the business activity support at run time, and to undertake work that might later be compensated by a compensation handler. If the application requires compensation when a business activity scope ends, the application passes the data that is required by the compensation process to a compensation handler indirectly, by using the business activity API. The data that is required by the compensation process can be in the form of either a serializable object or a Service Data Object (SDO).

Procedure

1. Design the application component that requires the business activity support. In particular, define the application component requirements for compensation and close activities. If the application component requires compensation, define the nature of the data in the serializable object or the SDO that the application component passes to the compensation handler.
2. Using the information from your application design, create the compensation handler for the application component, if required. This handler defines the close and compensation logic that runs upon completion of a business activity scope that has the handler added to it through an application component.
   a. Open your chosen WebSphere Application Server assembly tool.
   b. Create a new Java class that implements the appropriate interface, depending on the format of the data that is required by the compensation process:
      - For a serializable object, implement the com.ibm.websphere.wsba.serializable.CompensationHandler interface.
      - For an SDO, implement the com.ibm.websphere.wsba.CompensationHandler interface.

c. Implement the close and compensate methods on the new compensation handler object, to take appropriate actions depending on the serializable or SDO data that passes to the handler when it is invoked.

The compensation handler class is now ready for the application component to reference, and for assembly into an application.

3. Open the application component in the assembly tool.
4. Open the deployment descriptor for the application component in the deployment descriptor viewer.
5. Scroll to the **Compensation** section and select the **Run EJB methods under a Business Activity scope** check box.
6. In the **Compensation handler class** text field, type the fully qualified class name of the compensation handler class that you created earlier.
7. Save the deployment descriptor.
8. Build the application, including both the application component and the compensation handler. If the application is a web service, the application must be compliant with the Java Specification Request (JSR) 109 standard.
9. Deploy the application onto an application server that is business-activity-enabled.

Results

The application is now business-activity-enabled, and can use the business activity support at run time through the business activity API. The application component has a compensation handler associated with it, and can therefore call the setCompensationDataImmediate and setCompensationDataAtCommit methods at run time to add the compensation handler to the business activity scope. For more information about these methods, see the topics about the Business activity API. If the unit of work with which the business activity scope is associated fails, the compensation handler performs actions to compensate for the error.

What to do next

Ensure that the compensation handler class is on the application class path for the WebSphere Application Server runtime environment.

Business activity API

Use the business activity application programming interface (API) to create business activities and compensation handlers for an application component, and to log data that is required to compensate an activity if there is a failure in the overall business activity.

Overview

The business activity support provides a UserBusinessActivity API and two interfaces: a serializable.CompensationHandler interface and a CompensationHandler interface. Each interface has two exceptions: RetryCompensationHandlerException and CompensationHandlerFailedException. You can look up the UserBusinessActivity interface from the application server Java Naming and Directory Interface (JNDI) at `java:comp/websphere/UserBusinessActivity`. For example:

```
InitialContext ctx = new InitialContext();
UserBusinessActivity uba = (UserBusinessActivity) ctx.lookup("java:comp/websphere/UserBusinessActivity");
```

You can use the **`getId`** method to access the unique identifier for the business activity that is currently associated with the calling thread. The identifier is the same as the one that is generated for the business activity scope at run time and that is used for information, warning, and error messages. For example, the application can use the identifier in audit or diagnostic messages, and it is possible to correlate between application-generated and runtime-generated messages.

```
InitialContext initialContext = new InitialContext();
UserBusinessActivity uba = initialContext.lookup("java:comp/websphere/UserBusinessActivity");
...
String activityId = uba.getId();
if (activityId == null)
// No activity on the thread
else
// Output audit message including activity id
```

If an application component runs work that might require compensating upon failure in the business activity, you must provide a compensation handler class that is assembled as part of the deployed application. This Java class must implement one of the following interfaces:

- com.ibm.websphere.wsba.serializable.CompensationHandler, which takes a takes a parameter of a serializable object
- com.ibm.websphere.wsba.CompensationHandler, which takes a parameter of a Service Data Object (SDO)

Typically, applications that already have their data available in DataObject format will use the CompensationHandler interface, and applications that do not will use the serializable.CompensationHandler interface. Both interfaces support the close and compensate methods.

An application must register a compensation handler implementation that works with the type of compensation data (serializable object or SDO) that the application uses. If there is a mismatch between the type of data that the application component uses and the compensation handler implementation, there is an error.

During normal application processing, the application can make one or more invocations to the `setCompensationDataImmediate` or `setCompensationDataAtCommit` methods, passing in either a serializable object or an SDO that represents the current state of the work performed.

When the underlying unit of work (UOW) that the root business activity is associated with completes, all registered compensators are coordinated to complete. During completion, either the compensate or the close method is called on the compensation handler, passing in the most recent compensation data logged by the application component as a parameter. Your compensation handler implementation must be able to understand the data that is stored in either the serializable object or the SDO DataObject; when using this data, the compensation handler must be able to determine the nature of the work performed by the enterprise bean and compensate or close in an appropriate way, for example by undoing changes made to database rows if there is a failure in the business activity. You associate the compensation handler with an application component by using the assembly tooling, such as Rational Application Developer.

Active and inactive compensation handlers

You implement the serializable.CompensationHandler or CompensationHandler interface for any application component that executes code that might have to be compensated within a business activity scope. Compensation handler objects are registered implicitly with the business activity scope under which the application runs, whenever the application calls the UserBusinessActivity API to specify compensation data. Compensation handlers can be in one of two states, active or inactive, depending on any transactional UOW under which they are registered. A compensation handler that is registered within a transactional UOW might initially be inactive until the transaction commits, at which point the compensation handler becomes active (see the following section). A compensation handler that is registered outside a transactional UOW always becomes active immediately.

When a business activity completes, it drives only active compensation handlers. Any inactive compensation handlers that are associated with the business activity are discarded and never driven.

Logging compensation data

The business activity API specifies two methods that allow the application to log compensation data. This data is made available to the compensation handlers during their processing when the business activity completes. The application calls one of these methods, depending on whether it expects transactions to be part of the business activity.

setCompensationDataAtCommit()

Call the setCompensationDataAtCommit method when the application expects a global transaction on the thread.

- If a global transaction is present on the thread, the CompensationHandler object is initially inactive. If the global transaction fails, it rolls back any transactional work done within its transaction context in an atomic manner, and drives the business activity to compensate other completed UOWs. The compensation handler does not have to be involved. If the global transaction commits successfully, the compensation handler becomes active because if the overall business activity fails, the compensation handler is required to compensate the durable work that is completed by the global transaction. The setCompensationDataAtCommit method configures the CompensationHandler instance to undertake this compensation function.
- If a global transaction is not present when the setCompensationDataAtCommit method is called, the compensation handler becomes active immediately.

For example, for an SDO, and using the same business activity instance as in the previous example:

```
DataObject compensationData = doWorkWhichWouldNeedCompensating();
uba.setCompensationDataAtCommit(compensationData);
```

setCompensationDataImmediate()

Call the setCompensationDataImmediate method when the application does not expect a global transaction on the thread.

The setCompensationDataImmediate method makes a CompensationHandler instance active immediately, regardless of the current UOW context at the time that the method is invoked. The compensation handler is always able to participate during completion of the business activity.

The role of the setCompensationDataImmediate method is to compensate any non-transactional work, in other words, work that can be performed either inside or outside a global transaction, but that is not governed by the transaction. An example of this type of work is sending an email. The compensation handler must be active immediately so that if a failure occurs in a business activity, this non-transactional work is always compensated.

For example, for a serializable object, and using the same business activity instance as in the previous example:

```
Serializable compensationData = new MyCompensationData();
uba.setCompensationDataImmediate(compensationData);
```

Although these two compensation data logging methods, if called in the same enterprise bean, use the same compensation handler class, they create two separate instances of the compensation handler class at run time. Therefore, the actions of the methods are mutually exclusive; calling one of the methods does not overwrite any work carried out by the other method.

If a compensation handler instance is already added to the Business Activity by using one of these methods, and then the same method is called, passing in `null` as a parameter, that compensation handler instance is removed from the business activity, and is not driven to close or compensate during completion of the business activity.

As described previously, the business activity support adds a compensation handler instance to the business activity when a compensation data logging method is called for the first time by the enterprise bean that uses that business activity. At the same time, a snapshot of the enterprise application context is

taken and logged with the compensation data. When the business activity competes, all the compensation handlers that were added to the business activity are driven to compensate or close. The code that you create in the CompensationHandler or serializable.CompensationHandler class is guaranteed to run in the same enterprise application context that was captured in the earlier snapshot.

For details about the methods available in the business activity API, see the topic about additional APIs.

Chapter 36. Developing web services - Transports

Transport chains represent a network protocol stack that is used for I/O operations within an application server environment. Transport chains are part of the channel framework function that provides a common networking service for all components.

Configuring the SOAP over JMS transport for JAX-WS web services

SOAP over JMS protocol

The web services engine supports the use of an emerging industry standard SOAP over Java Message Service (JMS)-compliant messaging transport as an alternative to HTTP for communicating SOAP messages between clients and servers.

Note: This product supports an emerging industry standard SOAP over JMS protocol. The SOAP over JMS specification provides a standard set of interoperability guidelines for using a JMS-compliant transport with SOAP messages to enable interoperability between the implementations of different vendors. Using this standard, a mixture of client and server components from different vendors can interoperate when exchanging SOAP request and response messages over the JMS transport for both Java API for XML Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services. By using the JMS transport, your enterprise beans based web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections.

IBM and other vendors have been working on the proposed SOAP over JMS specification since 2005. The specification has been submitted to W3C and a working group is established. The current member submission of this draft specification was jointly published in October, 2007. Refer to the SOAP over JMS specification for details of this industry standard.

This topic provides a summary of the emerging industry-standard SOAP over JMS protocol. Use this SOAP over JMS transport protocol if you need to provide implementations for the client or server components. Also, you need to make sure that the implementations are interoperable with the client and server components provided by the web services engine in WebSphere Application Server.

The client component is responsible for sending SOAP request messages and receiving SOAP response messages while adhering to the following protocol constraints:

- The client must use a `javax.jms.BytesMessage` object or a `javax.jms.TextMessage` object to transmit the SOAP request message to the server.
- The client must set the following properties on the JMS request message before sending the message to the destination queue or topic:
  - **`SOAPJMS_contentType`**: This property is similar to the Content-Type header found in an HTTP message and is used to describe the content type of the message. A text-only SOAP message, for example, a message with no attachments, has the following value set for this JMS message property:

```
text/xml; charset="UTF-8"
```

    For a SOAP message containing attachments, use the following code to set the SOAPJMS_contentType property on the JMS message:

```
multipart/related; type="text/xml"; start="<...content-id_of_first_ part...>"
```

    This example represents a multipart message, where the first part is of type `text/xml` and contains the SOAP envelope. The other parts of the multipart message contain various attachments. The HTTP 1.1 specification contains more information about the Content-Type header.

- **enableTransaction**: Set this optional property to `true` on an outbound JMS request message if you want the server component to process the web service request under the same transaction that was used to receive the message from the destination queue or topic. This property is an IBM extension to the SOAP over JMS specification.

  Note: For client components, only set this property to `true` for one-way or two-way asynchronous requests to avoid synchronization problems that can occur with two-way synchronous web service requests. If this property is not set or is set to the default value of `false`, the server suspends the transaction that was used to receive the request message from the destination queue or topic prior to invoking the web services engine to process the request.
- **SOAPJMS_requestURI**: You must set this property to the JMS endpoint URL associated with the request.
- **SOAPJMS_soapAction**: This optional property is set on an outbound JMS request message to indicate the soapAction value associated with the web services request. This property is similar to the SOAPAction HTTP header used when transporting web service requests over an HTTP transport. The value of the soapAction property is a URI identifying the intent of the SOAP request. If the SOAPJMS_soapAction property is specified, it is used by the server component to determine the target of the request. The SOAP specification places no restrictions on the format or specificity of the URI nor does the specification require that the URI is resolvable. Typically, this property is set to the soapAction value from the WSDL document.
- **SOAPJMS_targetService**: You must set this property on an outbound JMS request message, and the value must match the targetService property value that is found in the JMS endpoint URL for the request. This value is used by the server component to determine the port component to which the request is dispatched.
- **SOAPJMS_bindingVersion**: This property indicates the version number of the protocol used by the client and server. Set the value to `1.0`.

- If the request message represents a two-way request, meaning that a reply is expected, the client component must set the JMS message JMSReplyTo property to specify the queue that is used for the reply message. The JMS message `setJMSReplyTo` method is used to specify the queue. You can benefit from configuring a permanent reply queue on the client to prevent the client from having to create a temporary queue each time a web service request is made. Read about configuring a permanent reply queue for web services using SOAP over JMS to learn more about creating this special queue.
- If the SOAP request message represents a one-way request, meaning that a reply message is not expected, the client component must not set the JMS message JMSReplyTo property.
- The client component can assume that a reply message is a JMS BytesMessage object.
- The client component can assume that the reply message correlation ID matches the message ID of the original request message.

The server component is responsible for receiving the SOAP request messages and sending the SOAP response messages, while adhering to the following protocol constraints:

- The server component can expect to receive a JMS BytesMessage. If something other than a BytesMessage is received by the server component, then a fault with the subcode, `unsupportedJMSMessageFormat`, is returned to the client if a reply is expected.
- The server component can expect to receive a `javax.jms.BytesMessage` object or a `javax.jms.TextMessage` object. If something other than a BytesMessage or TextMessage is received by the server component, then a fault with the subcode, `unsupportedJMSMessageFormat`, is returned to the client if a reply is expected.
- The server component must process the SOAP request properly to produce an appropriate SOAP reply message.
- The server component must send a reply message back to the client only if the JMS request message's JMSReplyTo property is set. The JMS message `getJMSReplyTo` method is used to retrieve the JMSReplyTo property value from the JMS message. This property value indicates the reply destination.

- When sending a reply message, the server component must use the same message type as the request. If the request was received as a BytesMessage, the reply must be sent as a BytesMessage. Similarly, if the request was received as a TextMessage, the reply must be sent as a TextMessage.
- The server component must set the following properties in the JMS reply message before sending the message to the reply queue:
  - **SOAPJMS_contentType**: This property is used to describe the content type of the message. See the description for this property in the client responsibilities section in this topic.
  - **correlation ID**: Set the correlation ID property of the JMS reply message to the message ID of the original JMS request message. This correlation is done by calling the JMS message `setJMSCorrelationID` method.
  - **SOAPJMS_bindingVersion**: This property indicates the version number of the protocol used by the client and server. Set the value to `1.0`.

The following example displays the results from calling the JMS message `toString` method for a request message without attachments:

```
  JMSMessage class: jms_bytes
  JMSType:          null
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     null
  JMSTimestamp:     -1
  JMSCorrelationID: null
  JMSDestination:   null
  JMSReplyTo:       queue://_Q_7D6C2035383215AB00000000000F4241?busName=WsFvtBus
  JMSRedelivered:   false
    JMS_IBM_MsgType: 1
    SOAPJMS_contentType: text/xml; charset=UTF-8
    SOAPJMS_targetService: MaelstromWsEndpoint
    SOAPJMS_requestIRI: jms:jndi:jms/MyRequestQueue?jndiConnectionFactoryName=jms/MyConnFactory&targetService=MyPort1
    SOAPJMS_soapAction: "getQuote"
    SOAPJMS_bindingVersion: 1.0
3c3f786d6c2076657273696f6e3d22312e302220656e636f64696e673d227574662d38223f3e3c73
6f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f7363
68656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c6e733a
7873643d22687474703a2f2f7777772e77332e6f72672f323030312f584d4c536368656d61222078
...
```

The following example displays the results from calling the JMS message `toString` method for a request message with attachments:

```
  JMSMessage class: jms_bytes
  JMSType:          null
  JMSDeliveryMode:  2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     null
  JMSTimestamp:     -1
  JMSCorrelationID: null
  JMSDestination:   null
  JMSReplyTo:       queue://_Q_F0940794C5CC2F84000000000044AA21?busName=WsFvtBus
  JMSRedelivered:   false
    JMS_IBM_MsgType: 1
    SOAPJMS_contentType: multipart/related;
  boundary=MIMEBoundaryurn_uuid_B6BAFEADB1886ADC241205525550237;
  type="text/xml"; start="<0.urn:uuid:B6BAFEADB1886ADC241205525550238@apache.org>"
    SOAPJMS_targetService: MaelstromWsEndpoint
    SOAPJMS_requestIRI: jms:jndi:jms/WebSvcsJMSQ?jndiConnectionFactoryName=
   jms/WebSvcsJMS_CF&targetService=MaelstromWsEndpoint
    SOAPJMS_soapAction: attachment
    SOAPJMS_bindingVersion: 1.0
2d2d4d494d45426f756e6461727975726e5f757569645f42364241464541444231383836414443332
343132303535323535353035303233370d0a436f6e74656e742d547970653a20746578742f786d6c3b20
636861727365743d5554462d380d0a436f6e74656e742d5472616e736665722d456e636f64696e67
3a20386269740d0a436f6e74656e742d49443a203c302e75726e3a757569643a4236424146454144
42313838364144433234313230353532353535303233384061706163686526e6f72673e0d0a0d0a3c
736f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f73
6368656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c6e73
3a7873643d22687474703a2f2f7777772e77332e6f72672f323030312f584d4c536368656d612220
786d6c6e733a7873693d22687474703a2f2f7777772e77332e6f72672f323030312f584d4c536368
656d612d696e7374616e63652220786d6c6e733a736f6170656e633d22687474703a2f2f73636865
...
```

The following example displays the results from calling the JMS message `toString` method for a SOAP reply message:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     null
JMSTimestamp:     0
JMSCorrelationID: ID:cdddb857f078a266eb9a972f110a134f0000000000000001
JMSDestination:   null
JMSReplyTo:       null
JMSRedelivered:   false
contentType:
  multipart/related;
  type="text/xml";
  start="<961368106530.1092112854745.IBM.WEBSERVICES@yackerjr>";
  boundary="----=_Part_0_1655006754.1092112854745"
0d0a2d2d2d2d2d2d3d5f506172745f305f313635353030363735342e313039323131323835343734
350d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d38
...
```

JMS endpoint URL syntax

As part of an emerging industry-standard SOAP over JMS protocol, a Java Message Service (JMS) endpoint URL syntax has been defined. A JMS endpoint URL is used to access Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) web services with the JMS transport. This URL specifies the JMS destination and connection factory, as well as the port component name for the Web service request. This endpoint URL is similar to the HTTP endpoint URL, which specifies the host and port as well as the context root and port component name.

Note: This product supports an emerging industry standard SOAP over JMS protocol. The SOAP over JMS specification provides a standard set of interoperability guidelines for using a JMS-compliant transport with SOAP messages to enable interoperability between the implementations of different vendors. Using this standard, a mixture of client and server components from different vendors can interoperate when exchanging SOAP request and response messages over the JMS transport for both Java API for XML Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services. By using the JMS transport, your enterprise beans based web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections.

IBM and other vendors have been working on the W3C SOAP over JMS specification since 2005. The specification has been submitted to W3C and a working group is established. The current member submission of this document was jointly published in October, 2007. The application server supports the current draft specification from W3C.

Note: A JMS endpoint URL has the following general form:

```
jms:jndi:<destination-jndi-name>?<property>=<value>&<property>=<value>&...
```

The URL consists of the `jms:` transport type, followed by the `jndi:` variant type, followed by the JNDI name of the destination queue or topic, followed by the query string containing a list of property and value pairs that are used to specify various JMS endpoint information. The `jndi:` variant means that JNDI is used to locate object names in the endpoint URL string.

The properties supported in the URL string are described in the following tables:

Table 240. Destination-related properties (required). Use these properties to specify destination-related properties for a JMS endpoint URL.

Property name	Description
jndiConnectionFactoryName	Specifies the JNDI name of the connection factory that is used by the client runtime to establish a connection to the JMS messaging engine.
targetService	Specifies the name of the port component to which the request is dispatched.

Table 241. JNDI-related properties (optional). Use these properties to specify JNDI-related properties for a JMS endpoint URL.

Property name	Description
jndiInitialContextFactory	Specifies the name of the initial context factory class to use. This value maps to the java.naming.factory.initial property.
jndiURL	Specifies the JNDI provider URL. This value maps to the java.naming.provider.url property.

Table 242. JMS-related properties (optional). Use these properties to specify JMS-related properties for a JMS endpoint URL.

Property name	Description
deliveryMode	Indicates whether the request message is persistent or not. The valid values are `PERSISTENT` and `NON_PERSISTENT`. The default value is `NON_PERSISTENT`.
timeToLive	Specifies the lifetime, in milliseconds, of the request message. A value of `0` indicates an infinite lifetime. If this parameter is not specified, then the JMS-defined default value is used.
priority	Specifies the JMS priority associated with the request message. Specify this value as a positive integer from 0, the lowest priority, to 9, the highest priority. If this parameter is not specified, then the JMS-defined default value is used.
replyToName	Specifies the JNDI name of the JMS destination to which the response message is sent. Using this optional property enables the client to use a previously defined, permanent queue rather than a temporary queue, for receiving replies.
messageType	Specifies the message type to use with the request message. A value of `BYTES` indicates the javax.jms.BytesMessage object is used. A value of `TEXT` indicates javax.jms.TextMessage object is used. The default value is `BYTES`.

The required properties jndiConnectionFactoryName and targetService must be in the JMS endpoint URL string. The remaining properties are optional.

If you set values for the deliveryMode, timeToLive, and priority properties on the JMS request, these values are propagated from the JMS request message to the corresponding JMS reply message.

See the SOAP over Java Message Service specification in the web services specifications and APIs documentation to learn more about this industry standard.

IBM proprietary SOAP over JMS protocol (deprecated)

You can use a SOAP over Java Message Service (JMS) transport as an alternative to HTTP for communicating SOAP messages between clients and servers. The web services engine supports the use of an IBM proprietary implementation as well as the industry standard implementation.

Note: In earlier versions of the application server, an IBM proprietary SOAP over JMS protocol was supported for Java API for XML-based RPC (JAX-RPC) applications. In WebSphere Application Server 7.0 and later, this proprietary SOAP over JMS protocol is now deprecated in favor of an emerging industry standard SOAP over JMS protocol. You can use the IBM proprietary SOAP over JMS protocol with your Java API for XML Web Services (JAX-WS) or JAX-RPC web services; however, take advantage of the emerging standard SOAP over JMS protocol. If your client application invokes enterprise beans-based web services that are supported by an earlier version of the WebSphere Application Server, you must continue to use the IBM proprietary SOAP over JMS protocol to access those web services.

You can use a SOAP over JMS transport if you need to provide implementations for the client or server components, and you need to make sure that the implementations are interoperable with the client and server components provided by the web services engine in the application server. The IBM proprietary SOAP over JMS protocol describes specific message exchange requirements for client and server components so they can exchange SOAP request and response messages through the use of the JMS APIs supported by the application server.

The client component is responsible for sending SOAP request messages and receiving SOAP response messages while adhering to the following protocol constraints:

- The client must use either a JMS TextMessage object, for example, javax.jms.TextMessage, or a BytesMessage object, for example, javax.jms.BytesMessage, to transmit the SOAP request message to the server. If the request message contains attachments, a BytesMessage object must be used. If the request message does not contain attachments, the client can use a TextMessage or a BytesMessage object. The application server client implementation uses only a BytesMessage object for the request message due to the potential need to transmit attachments.
- The client must set the following properties on the JMS request message before sending the message to the destination queue or topic:
  - contentType: This property is similar to the Content-Type header found in an HTTP message and is used to describe the content type of the message. A text-only SOAP message, for example, a message with no attachments, is written as follows:

```
text/xml; charset="UTF-8"
```

    The **contentType** property in a SOAP request message that contains attachments must be set as follows:

```
multipart/related; type="text/xml"; start="<...content-id of first part...>"
```

    This example represents a multi-part message, where the first part is of type `text/xml` that contains the SOAP message. The other parts of the multi-part message contain various attachments. The HTTP 1.1 specification contains more information about the Content-Type header.
  - enableTransaction: Set this optional property to `true` on the outgoing SOAP over JMS request message if the server component should process the web service request under the same transaction that was used to receive the message from the destination queue or topic. The client component should only set this property to `true` for a one-way request to avoid synchronization problems that can occur with a two-way web service request. If this property is not set or is set to the default value of `false`, then the server will suspend the transaction that was used to receive the request message from the destination queue or topic prior to invoking the web services engine to process the request.

    gotcha: The client component should only set this property to `true` for a one-way request. Setting this property to `true` for a two-way web server request is not supported because of synchronization problems.
  - endpointURL: This property must be set to the JMS endpoint URL associated with the request.
  - soapAction: This optional property is set on an outgoing SOAP over JMS request message to indicate the soapAction value associated with the web services request. This property is similar to the SOAPAction HTTP header used when transporting web service requests over an HTTP transport. The value of the soapAction property is a URI identifying the intent of the SOAP request. If the soapAction property is specified, it is used by the server component to determine the target of the request. The SOAP specification places no restrictions on the format or specificity of the URI or that it is resolvable. Typically, this property is set to the soapAction value from the WSDL document.
  - targetService: This property must be set to the targetService property value that is found in the JMS-style endpoint location URL for the request. This value is used by the server component to determine the port component in the target when dispatching the request.
  - transportVersion: This property indicates the version number of the protocol used by the client and server. Set the value to 1 (one).
- If the SOAP request message represents a two-way request, the client component must set the JMS message's replyTo property to specify the queue that is used for the reply message. The JMS message `setJMSReplyTo` method is used for this. It can be beneficial to configure a permanent **replyTo** queue on the client to prevent the client from having to set the JMS message's replyTo property each time a web service request is made.

- If the SOAP request message represents a one-way request, the client component must not set the JMS message's replyTo property.
- The client component must be prepared to handle a reply message that is a BytesMessage or a TextMessage object, regardless of the type of JMS message used to transmit the SOAP request. The application server component responds with the same type of JMS message that is received from the client, unless the response contains attachments and a BytesMessage object must be used.
- The client component can assume that the reply message correlation ID matches the original request message ID.

The server component is responsible for receiving the SOAP request messages and sending the SOAP response messages while adhering to the following protocol constraints:

- The server must be prepared to receive a TextMessage or a BytesMessage. If the request contains attachments, a ByteMessage must be used. The WebSphere product implementation of the server component responds in kind when sending the reply message back to the client, unless the response contains attachments and a BytesMessage is used.
- The server component must process the SOAP request properly to produce an appropriate SOAP reply message.
- The server component must send a reply message back to the client only if the JMS request message's replyTo property is set.
- The server component must set the following properties in the JMS reply message before sending the message to the replyTo queue:
  - contentType: See the description for this property in the client responsibilities section in this article.
  - Set the **correlation ID** of the JMS reply message to the message ID of the original JMS request message. This is done by calling the JMS message `setJMSCorrelationID` method.
  - transportVersion: This property indicates the version number of the protocol used by the client and server. Set the value to 1 (one).

The following example displays the results from calling the JMS message `toString` method for a request message without attachments:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     ID:d438eebf04cb124aa25c5821110a134f0000000000000001
JMSTimestamp:     1092110476167
JMSCorrelationID: null
JMSDestination:   topic://NewsGroupTopic?topicSpace=FvtTopicSpace
JMSReplyTo:       null
JMSRedelivered:   false
JMS_IBM_System_MessageID: 6B6765B36943A18C_11000001
transportVersion: 1
JMSXUserID:
targetService: NGConsumerJMS
JMSXAppID: Service Integration Bus
endpointURL: jms:/topic?destination=jms/NewsGroupTopic&connectionFactory;
  =jms/NewsGroupTCF&targetService;=NGConsumerJMS

contentType: text/xml; charset=utf-8
3c736f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f
736368656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c6e
...
```

The following SOAP Version 1.1 example displays the payload from the previous message example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<postMessage><ngName xsi:type="xsd:string">news.current.events</ngName>
```

```
<msg xsi:type="xsd:string">This is a sample news item.</msg>
</postMessage>
</soapenv:Body>
</soapenv:Envelope>
```

For SOAP Version 1.2, the `encodingStyle` parameter is not supported, so the example looks similar to the following:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<postMessage><ngName xsi:type="xsd:string">news.current.events</ngName>
<msg xsi:type="xsd:string">This is a sample news item.</msg>
</postMessage>
</soapenv:Body>
</soapenv:Envelope>
```

The following example displays the results from calling the JMS message `toString` method for a request message with attachments:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  1
JMSExpiration:    1092086312310
JMSPriority:      4
JMSMessageID:     ID:4bb64ed64e7d813d59ba5fec110a134f0000000000000001
JMSTimestamp:     1092086012310
JMSCorrelationID: null
JMSDestination:   queue://Logger_Q
JMSReplyTo:       queue://_Q_6B6765B36943A18C_00000385
JMSRedelivered:   false
JMS_IBM_System_MessageID: 6B6765B36943A18C_10000001
transportVersion: 1
JMSXUserID:
targetService: WSLoggerJMS
JMSXAppID: Service Integration Bus
endpointURL: jms:/queue?
destination=jms/Logger_Q&connectionFactory=jms/Logger_CF&targetService=WSLoggerJMS
contentType: multipart/related; type="text/xml";
start="<945414389.1092086011970.IBM.WEBSERVICES@myhost1>";
boundary="----=_Part_0_247953397.1092086011970"
0d0a2d2d2d2d2d2d3d5f506172745f305f323437393533333939372e313039323038363031313937330
0d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d380d
...
```

The following displays the payload from the previous message example:

```
Content-Type: multipart/related; type="text/xml";

       start="<945414389.1092086011970.IBM.WEBSERVICES@myhost1>";

       boundary="----=_Part_0_247953397.1092086011970"



------=_Part_0_247953397.1092086011970
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <945414389.1092086011970.IBM.WEBSERVICES@myhost1>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header>
<p499:InternationalizationContext soapenv:mustUnderstand="0"
 xmlns:p499="http://www.ibm.com/webservices/InternationalizationContext">
<Locales>
 <Locale>
  <LanguageCode>en</LanguageCode>
  <CountryCode>US</CountryCode>
 </Locale>
</Locales>
<TimeZoneId>America/Chicago</TimeZoneId>
</p499:InternationalizationContext>
</soapenv:Header>
```

```
<soapenv:Body>
 <sendJpegImage/>
</soapenv:Body>
<soapenv:Envelope>
------=_Part_0_247953397.1092086011970
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <jpegImageRequest=81380956150.1092086011880.IBM.WEBSERVICES@myhost1>
<...contents of jpeg image file...>
```

The following example displays the results from calling the JMS message `toString` method for a SOAP reply message:

```
JMSMessage class: jms_bytes
JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     null
JMSTimestamp:     0
JMSCorrelationID: ID:cdddb857f078a266eb9a972f110a134f0000000000000001
JMSDestination:   null
JMSReplyTo:       null
JMSRedelivered:   false
contentType:
  multipart/related;
  type="text/xml";
  start="<961368106530.1092112854745.IBM.WEBSERVICES@yackerjr>";
  boundary="----=_Part_0_1655006754.1092112854745"
0d0a2d2d2d2d2d2d3d5f506172745f305f313635353030363735342e313039323131323835343734
350d0a436f6e74656e742d547970653a20746578742f786d6c3b20636861727365743d5554462d38
...
```

IBM proprietary JMS endpoint URL syntax (deprecated)

A Java Message Service (JMS) endpoint URL is used to access Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) web services with the JMS transport. This proprietary URL specifies the Java Message Service (JMS) destination and connection factory, as well as the port component name for the web service request. This endpoint URL is similar to the HTTP endpoint URL, which specifies the host and port as well as the context root and port component name.

Note: A JMS endpoint URL has the following general form:

`jms:/[queue|topic]?<property>=<value>&<property>=<value>&...`

> The URL consists of the `jms:` transport type, followed by either `/queue` or `/topic` to indicate the JMS destination type, followed by the query string containing a list of property and value pairs that are used to specify the JMS endpoint information.

The properties supported in the URL string are described in the following tables:

Table 243. Destination-related properties (required). Use these properties to specify destination-related properties for a JMS endpoint URL.

Property name	Description
destination	Specifies the Java Naming and Directory Interface (JNDI) name of the destination queue or topic.
connectionFactory	Specifies the JNDI name of the connection factory.
targetService	Specifies the name of the port component to which the request is dispatched.

Table 244. JNDI-related properties (optional). Use these properties to specify JNDI-related properties for a JMS endpoint URL.

Property name	Description
initialContextFactory	Specifies the name of the initial context factory to use which is mapped to the java.naming.factory.initial property.
jndiProviderURL	Specifies the JNDI provider URL, which is mapped to the java.naming.provider.url property.

Table 245. JMS-related properties (optional). Use these properties to specify JMS-related properties for a JMS endpoint URL.

Property name	Description
deliveryMode	Indicates whether the request message is persistent or not. The valid values are `1` for nonpersistent and `2` for persistent. The default value is `1`.
timeToLive	Specifies, in milliseconds, the lifetime of the JMS request message. The default value of `0` indicates an infinite lifetime. However, when you are using web services, this property is bounded by the value that is specified for the synchTimeout property. The setting for the synchTimeout property determines how long a client waits for a response from the server. This bounding prevents the JMS message from remaining active after the client has stopped waiting for a response from the server.
priority	Specifies the JMS priority associated with the request message. Valid values are between `0` to `9`. The default value is `4`. A value of `0` is the lowest priority and a value of `9` is the highest priority.
replyToDestination	Specifies the JNDI name of a queue to be used to receive reply messages. Using this optional property enables the client to use a permanent queue, rather than a temporary queue, for receiving replies.

If you set values for the deliveryMode, timeToLive, and priority properties on the JMS request, these values are propagated from the JMS request message to the corresponding JMS reply message.

The required properties, destination, connectionFactory, and targetService must be contained in the JMS endpoint URL string. The rest of the properties are optional.

You can set any of the properties on the client Stub object. The various properties can be specified by including them as part of the endpoint URL or you can set these properties programmatically by the client on the Stub object. Properties specified on the client Stub object take precedence over properties that are specified as part of a JMS endpoint URL string.

Invoking web service requests transactionally using SOAP over JMS transport

Use the `enableTransactionalOneWay` property to ensure that one-way and two-way asynchronous web service requests using the industry standard SOAP over JMS transport will be sent to the destination queue or topic transactionally.

About this task

When using JMS to transport Java API for XML Web Services (JAX-WS) or Java API for XML-based RPC (JAX-RPC) web service requests, the default behavior is for the SOAP message to be added to the destination queue or topic non-transactionally or outside of the client application's transaction. Adding the SOAP message to the destination queue or topic is done outside of the transaction to avoid synchronization problems that can occur with two-way synchronous web service requests. However, you can choose to enable one-way and two-way asynchronous requests to be processed as part of a transaction. You can use the `enableTransactionalOneWay` property to ensure that one-way and two-way asynchronous web service requests that use the JMS transport are sent to the destination queue or topic transactionally. When the client application invokes the web service request, the resulting SOAP request message is added to the destination queue or topic as part of the client application's transaction.

Use one of the following ways to enable the `enableTransactionalOneWay` property.

Procedure

- Set the `enableTransactionalOneWay` property programmatically. The value of the property is a Boolean.
  - For JAX-WS clients, set the property on the client JAX-WS `RequestContext` object. For example:

```
((BindingProvider) port).getRequestContext().put
(com.ibm.websphere.webservices.Constants.ENABLE_TRAN_ONEWAY,
new Boolean(true));
```

  - For JAX-RPC clients, set the property on the client JAX-RPC Stub or Call object. For example:

```
stub._setProperty(com.ibm.websphere.webservices.Constants.ENABLE_TRAN_ONEWAY,
new Boolean(true));
```

- For JAX-RPC clients, set the `enableTransactionalOneWay` property as a custom property in the `ibm-webservicesclient-bnd.xmi` deployment descriptor file by using the **wsadmin** command.

  For more information about the wsadmin tool options, see the options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands information..

  Use the `$AdminApp` object along with the `-WebServicesClientCustomProperty` option to set the value of the property within the client binding file, `ibm-webservicesclient-bnd.xmi`. The value of the custom property, `enableTransactionalOneWay`, is either `true` or `false`.
  - Using Jacl:

    ```
    $AdminApp edit MyApplication {-WebServicesClientCustomProperty {{MyEJBJar.jar MyEJB
    service/MyServiceRef MyPort enableTransactionalOneWay true}}}
    ```
  - Using Jython:

    ```
     AdminApp.edit('MyApplication', ['-WebServicesClientCustomProperty', [['MyEJBJar.jar',
    'MyEJB', 'service/MyServiceRef ', 'MyPort', 'enableTransactionalOneWay', 'true']]])
    ```

Results

You have a web services client application that is configured to invoke one-way and two-way asynchronous requests transactionally when using the JMS transport.

What to do next

After you have enabled the `enableTransactionalOneWay` property, run the client application.

Invoking one-way JAX-RPC web service requests transactionally using the JMS transport (deprecated)

Use the `enableTransactionalOneWay` property to ensure that one-way JAX-RPC web service requests using the IBM proprietary JMS transport will be sent to the destination queue or topic transactionally.

About this task

Note: Beginning with WebSphere Application Server 7.0, the IBM proprietary SOAP over JMS protocol is deprecated in favor of the emerging industry standard protocol. You can use the IBM proprietary SOAP over JMS protocol with your Java API for XML Web Services (JAX-WS) or JAX-RPC web services, however, you are encouraged to take advantage of the SOAP over JMS protocol standard. This task describes configuring a permanent replyTo queue when using the IBM proprietary SOAP over JMS transport. To learn more about the SOAP over JMS standard, see the using SOAP over JMS to transport web services documentation.

When using JMS to transport web service requests, the default behavior is for the SOAP message to be added to the destination queue or topic non-transactionally or outside of the client application's transaction. Adding the SOAP message to the destination queue or topic is done outside of the transaction to avoid synchronization problems that can occur with two-way web service requests. However, you can choose to enable one-way requests to be processed as part of the transaction. The `enableTransactionalOneWay` property can be used to ensure that one-way web service requests that use the JMS transport will be sent to the destination queue or topic transactionally. When the client application invokes the one-way web service request, the resulting SOAP request message is added to the destination queue or topic as part of the client application's transaction.

Use one of the following ways to enable the `enableTransactionalOneWay` property.

Procedure

- Set the `enableTransactionalOneWay` property programmatically on the client JAX-RPC Stub or Call object.

When using a static Stub to invoke the web service operation, set the `enableTransactionalOneWay` property on the Stub object before invoking the web service method. When using a Call object to invoke the web service operation, set the `enableTransactionalOneWay` property on the Call object before invoking the `invokeOneWay()` method.

```
Service service = /* Obtain the desired service */
MyStub stub = service.getPort();

/* Set enableTransactionalOneWay property on Stub */
stub._setProperty(com.ibm.websphere.webservices.Constants.ENABLE_TRAN_ONEWAY, new Boolean(true));

/* Invoke the one-way operation */
stub.myOneWayOperation("Parm1");
```

The value of the property is Boolean.

- Set the `enableTransactionalOneWay` property as a custom property in the `ibm-webservicesclient-bnd.xmi` deployment descriptor file by using the **wsadmin** command.

  For more information about the wsadmin tool options, see the options for the AdminApp object install, installInteractive, edit, editInteractive, update, and updateInteractive commands information.

  Use the `$AdminApp` object along with the `-WebServicesClientCustomProperty` option to set the value of the property within the client binding file, `ibm-webservicesclient-bnd.xmi`. The value of the custom property, `enableTransactionalOneWay`, is either `true` or `false`.
  - Using Jacl:

    ```
    $AdminApp edit MyApplication {-WebServicesClientCustomProperty {{MyEJBJar.jar MyEJB
    service/MyServiceRef MyPort enableTransactionalOneWay true}}}
    ```
  - Using Jython:

    ```
     AdminApp.edit('MyApplication', ['-WebServicesClientCustomProperty', [['MyEJBJar.jar',
    'MyEJB', 'service/MyServiceRef ', 'MyPort', 'enableTransactionalOneWay', 'true']]])
    ```

Results

You have a web service client application that is configured to invoke one-way requests transactionally while using the JMS transport.

What to do next

After you have enabled the `enableTransactionalOneWay` property, run the client application.

Configuring SOAP over JMS message types

You can configure your SOAP over Java Message Service (JMS) request or response messages to use either BytesMessage or TextMessage objects.

Before you begin

A web service must be implemented as an enterprise bean for accessibility through the JMS transport.

About this task

For your web services that use the emerging industry standard SOAP over JMS protocol, you can configure the SOAP over JMS requests and responses to specify whether the messages are transmitted within JMS BytesMessage (`javax.jms.BytesMessage`) or TextMessage (`javax.jms.TextMessage`) objects. The default message type is BytesMessage.

If the JMS message is a BytesMessage, the body of the JMS message is binary data.

If the JMS message is a TextMessage, the body of the JMS message is string data. For example, consider configuring text messages if you want to implement an audit or logging facility that requires JMS messages that are human readable.

When using SOAP over JMS as a transport for web services request and response messages, it is important to understand the following performance considerations when deciding whether to use BytesMessage or TextMessage objects:

- Memory usage -

  The payload within a TextMessage is string-based and because the individual characters are based on the UTF-16 character encoding, the payload within a TextMessage is likely to occupy twice as many bytes as an equivalent BytesMessage. For request and response messages with small payloads, this difference in memory usage between TextMessages and BytesMessages might not be important, but for large payloads, the difference in memory usage might become more important to consider for your business environment.

- Binary attachments -

  When a JMS TextMessage is used to transport request and response messages, any binary attachments that are included along with the SOAP message body must be base64-encoded because the underlying message payload is string-based. When binary attachments are base64-encoded, additional processing time and memory are required to perform the transformation on that data. As a result, it is possible that the base64-encoded binary attachment might be up to one-third larger than the original unencoded version of the attachment. In this case, if this larger encoded attachment part is then transported in a string-based message payload, the memory usage required to transport the binary attachment part might be almost three times the memory size of the original unencoded binary attachment within a BytesMessage.

Procedure

- Specify the JMS message type for JAX-WS applications.

  You can configure the JMS message type to specify `BYTES` or `TEXT` in the following ways when using JAX-WS applications. The JAX-WS Web services engine searches for the messageType property in the order of this list and uses the first setting that it finds.

  – Set the messageType property in the JMS endpoint location URL string; for example:

```
jms:jndi:jms/MyQueue&....&messageType=TEXT
```

    See the JMS endpoint URL syntax information for details on how to set this property using this method.

  – Set the com.ibm.websphere.webservices.transport.jms.messageType custom property in the JMS transport policy binding in the administrative console. See the JMS transport bindings information for details on how to set this property using the administrative console.

  – Set the com.ibm.websphere.webservices.transport.jms.messageType property programmatically on the RequestContext object of the JAX-WS BindingProvider.

  – Set the com.ibm.websphere.webservices.transport.jms.messageType global system property in the JVM process under which the application server is running. See the Java virtual machine custom properties information for details on how to set this property using this method.

- Specify the JMS message type for JAX-RPC applications.

  You can configure the JMS message type to specify `BYTES` or `TEXT` in the following ways when using JAX-RPC applications. The JAX-RPC Web services engine searches for the messageType property in the order of this list and uses the first setting that it finds.

  – Set the messageType property in the JMS endpoint location URL string; for example:

```
jms:jndi:jms/MyQueue&....&messageType=TEXT
```

    See the JMS endpoint URL syntax information for details on how to set this property using this method.

  – Set the com.ibm.websphere.webservices.transport.jms.messageType property programmatically on the Stub or Call object of the client application..

  – Set the com.ibm.websphere.webservices.transport.jms.messageType custom property in the client binding file, META-INF/ibm-webservicesclient-bnd.xmi.

– Set the com.ibm.websphere.webservices.transport.jms.messageType global system property in the JVM process under which the application server is running. See the Java virtual machine custom properties information for details on how to set this property using this method.

Results

You have configured a web service client to use either TextMessage or BytesMessage objects when using the SOAP over JMS protocol to transmit request and response messages.

Chapter 37. Developing web services - UDDI registry

The Universal Description, Discovery, and Integration (UDDI) specification defines a way to publish and discover information about web services. The UDDI specification defines a standard for the visibility, reusability, and manageability that are essential for a service-oriented architecture (SOA) registry service. The UDDI registry is a directory for web services that is implemented using the UDDI specification. It is a component of WebSphere® Application Server.

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. They implement a services oriented architecture (SOA), which supports the connecting or sharing of resources and data in a very flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

Developing with the UDDI registry

You can access the UDDI registry in several ways; the UDDI registry user interface, application programming interfaces (APIs), or the Java API for XML Registries (JAXR).

About this task

You can access the UDDI registry programmatically by using several application programming interfaces (APIs).

You can also explore the UDDI registry by using the UDDI registry user interface (also referred to as the UDDI registry user console), which is a graphical interface.

You can access both UDDI (Version 2 only) and ebXML registries by using the Java API for XML Registries (JAXR). This Java client API is part of the Java EE specification.

Procedure

- "UDDI registry client programming"
- "Using the UDDI registry user interface" on page 1792
- "Using the JAXR provider for UDDI" on page 1798

UDDI registry client programming

The UDDI registry provides several application programming interfaces (APIs) that you can use to access the UDDI registry programmatically.

About this task

The UDDI Version 3 registry supports multiple versions of UDDI. It supports UDDI Version 1, Version 2, and Version 3.

For details of the Version 1 and Version 2 API, refer to the UDDI Version 2 Specifications.

For details of the UDDI Version 3.0.2 API, refer to the UDDI Version 3.0.2 Specification.

The UDDI registry information in this information center defines the support that the UDDI registry provides for the UDDI Version 3.0.2 specification and associated addenda.

The following UDDI Version 3 API sets are supported:
- The UDDI V3 Inquiry API
- The UDDI V3 Publish API
- The UDDI V3 Custody and Ownership Transfer API

- The UDDI V3 Security API

Restriction: In DB2 for zSeries® Version 7, the length of publish and inquiry strings are limited to 255 characters. If this limit is exceeded, error 10500 (E_Fatal) is returned. If you use a character set that uses multiple byte characters, it is easy to exceed this limit. Therefore, use care if you use this type of character set.

Procedure

1. Learn about the standard aspects of the UDDI APIs by using the following topics.
   - "UDDI registry Version 3 entity keys" on page 1784 explains UDDI entity keys, and the capability with UDDI Version 3 to save UDDI entities with publisher-assigned keys.
   - "Digital signatures and the UDDI registry" on page 1787 explains the support for digital signing of UDDI entities, and for validation of signatures.
2. Access the APIs programmatically. The recommended client API is the UDDI Version 3 Client, which allows access to the UDDI Version 3 APIs from Java client code.

   Other client APIs are provided for compatibility with previous versions of the UDDI registry:
   - The UDDI4J programming interface provides Java class libraries for accessing UDDI Version 1 and Version 2 APIs. These class libraries are both deprecated in this release, and are replaced by the UDDI Version 3 Client for Java.
   - The UDDI EJB Interface provides an Enterprise JavaBeans (EJB) interface to the UDDI Version 2 APIs. The UDDI EJB interface is deprecated in this release.

   Although the recommended programmatic access to the UDDI APIs is through the UDDI Version 3 Client for Java, it is also valid to use the UDDI APIs directly by using SOAP. To use the SOAP API, construct a properly-formed UDDI message in the body of a SOAP request, and send it by using HTTP POST to the appropriate SOAP endpoint for the UDDI service. The response is returned in the body of the HTTP reply.

   The UDDI registry samples include samples that demonstrate how to program directly to the SOAP API. The samples are written in Java code, but you can use other programming languages to create your SOAP client, provided that you still send requests that are compliant to the SOAP specification. Valid UDDI requests must conform to the UDDI schema, as detailed in the UDDI specification

   Support is also provided to use HTTP GET to return XML representations of UDDI entities: see "HTTP GET services for UDDI registry data structures" on page 1788 for details.

Inquiry API for the UDDI Version 3 registry

The Inquiry API provides four forms of query that follow broadly used conventions that match the needs of software that is traditionally used in registries.

The four forms of query are:
- The browse pattern
- The drilldown pattern
- The invocation pattern
- Inquiry API functions

Browse pattern for the UDDI registry

Software that is used to explore and examine data, especially hierarchical data, requires browse capabilities. The browse pattern characteristically involves starting with some broad information, performing a search, finding general result sets, then selecting more specific information for drill-down patterns.

The UDDI API specifications accommodate the browse pattern with the *find_xx* API calls. These calls form the search capabilities that the API provides. The calls are matched with summary return messages that return overview information about the registered information that is associated with the inquiry message type and the search criteria specified in the inquiry.

A typical browse sequence might involve finding whether there is any information registered for a business you know about. This sequence starts with a call to find_business, perhaps passing the first characters of the business name that you know. This action returns a businessList result. This result is overview information, including keys, names, and descriptions, that is derived from the registered businessEntity information, matching on the name fragment that you provide. If the business you are looking for is in this list, you can use the find_service API call to drill down into the corresponding businessService information, and look for specific technical models, such as purchasing or shipping. Similarly, if you know the technical *fingerprint*, that is, the tModel signature, of a particular software interface, and you want to see if the business you are looking for provides a web service that supports that interface, you can use the find_binding inquiry message.

Drilldown pattern for the UDDI registry

When you have a key for one of the four main data types that a UDDI registry manages, you can use that key to access the full registered details of a specific data instance. The UDDI data types are businessEntity, businessService, bindingTemplate, and tModel. You can access the full registered information for any of these structures by passing a relevant key type to one of the *get_xx* API calls.

Continuing the example from the previous section, one data item that is returned by all of the *find_x* return sets is key information. For the business you are interested in, the businessKey value that is returned in the contents of a businessList structure can be passed as an argument to the get_businessDetail message. The successful return to this message is a businessDetail message that contains the full registered information for the entity with the key value that is passed. This information will be a full businessEntity structure.

Invocation pattern for the UDDI registry

For an application to take advantage of a remote web service that is registered in the UDDI registry by other businesses or entities, you must prepare that application to use the information found in the registry for the specific service being invoked.

The *bindingTemplate* data that is obtained from the UDDI registry represents the specific details about an instance of a given interface type, including the location at which a program starts interacting with the service. The calling application or program caches this information and uses it to contact the service at the registered address whenever the calling application needs to communicate with the service instance.

In remote procedure technologies that were previously popular, tools automate the tasks that are associated with caching, or hard coding, location information. However, there are problems when a remote service moves and the callers do not know about the move. There are many reasons why a remote service might move, for example, a server upgrade, disaster recovery, service acquisition, or a change to the business name.

When a call fails using cached information previously obtained from a UDDI registry, the correct behavior is to query the UDDI registry for fresh bindingTemplate information. If the data that is returned is different from the cached information, the service invocation can automatically try the invocation again, using the fresh information. If the result of this retry is successful, the new information replaces the cached information.

By using this pattern with web services, a business that uses a UDDI registry can automate the recovery of a large number of partners without unneccessary communication and coordination costs. For example, if a business activates a disaster recovery site, most of the calls from partners fail when they try to invoke services at the failed site. By updating the UDDI information with the new address for the service, partners who use the invocation pattern automatically locate the new service information and recover without further administrative action.

Inquiry API functions in the UDDI registry:

You can use the Inquiry API set to locate and obtain details about entries in a UDDI registry.

The Inquiry API is split into a number of functions, where each function requires some mandatory and some optional arguments.

The find_*xx* API functions can accept an optional findQualifiers argument.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java. To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Inquiry API calls:

find_binding
: Locates specific bindings in a registered businessService. Returns a bindingDetail message that contains zero or more bindingTemplate structures that match the criteria specified in the argument list.

find_business
: Locates information about one or more businesses. Returns a businessList message that matches the conditions specified in the arguments.

find_relatedBusinesses
: Locates information about businessEntity registrations that are related to a specific business entity whose key is passed in the inquiry. The related businesses feature is used to manage registration of business units and subsequently relate them based on organizational hierarchies or business partner relationships. Returns a relatedBusinessList message that contains results that match the conditions specified in the argument list.

find_service
: Locates specific services in a registered businessEntity. Returns a serviceList message that matches the conditions specified in the arguments.

find_tModel
: Locates a list of tModel entities that match a set of specified criteria. The response is a list of abbreviated information about registered tModel data that matches the specified criteria. The result is returned in a tModelList message.

get_bindingDetail
: Requests the runtime bindingTemplate information for the purpose of invoking a registered business API. Returns a bindingDetail message.

get_businessDetail
: Returns complete businessEntity information for one or more specified businessEntity registrations that match the specified businessKey values. Returns a businessDetail message.

get_opertionalInfo
: Gets full operational information pertaining to one or more entities in the registry. Returns an operationalInfos structure.

get_serviceDetail
: Requests full information about a known businessService structure. Returns a serviceDetail message.

get_tModelDetail
: Gets full details for a given set of registered tModel data. Returns a tModelDetail message.

For details of the query syntax, refer to the UDDI Version 3 API specification.

FindQualifier values for API functions in the UDDI registry:

The find_*xx* API functions (find_business, find_service, find_binding, find_tModel and find_relatedBusinesses) accept an optional findQualifiers argument, which can contain multiple findQualifier values.

The following list contains the findQualifier short names, a brief description, and the appropriate find function.

andAllKeys
Specifies that the identifierBag element uses AND behavior with keys, rather than OR behavior. This behavior is the default for the categoryBag and tModelBag elements. This value applies to the find_business, find_service, find_binding, and find_tModel functions. This value does not apply to the find_relatedBusinesses function.

approximateMatch
Specifies that wildcard search behavior is required. This is no longer the default behavior; the default behavior is specified by the exactMatch value. This value applies to the find_business, find_service, find_binding, find_tModel, and find_relatedBusiness functions.

binarySort
Specifies a faster sort by using a binary sort by name, as represented in Unicode codepoints. This value applies only to the find_business, find_service and find_tModel functions.

bindingSubset
This value is used only with a categoryBag element in the find_business or find_services functions.

caseInsensitiveMatch
Specifies that the matching behavior for name, keyValue, and keyName, where applicable, is not case-sensitive. By default, the matching behavior is case-sensitive. This value applies to the find_business, find_service, and find_tModel functions.

caseInsensitiveSort
Specifies that the sorting behavior for name, keyValue, and keyName, where applicable, is not case-sensitive. By default, the sorting behavior is case-sensitive.

caseSensitiveMatch
Specifies that the matching behavior for name, keyValue, and keyName, where applicable, is case-sensitive. This is the default behavior. This value applies to the find_business, find_service, find_binding, find_tModel, and find_relatedBusinesses functions.

caseSensitiveSort
Specifies that the sorting behavior for the result set is case-sensitive. This is the default behavior. This value applies to the find_business, find_service, and find_tModel functions.

combineCategoryBags
For a find_business function, specifies that the categoryBag entries for the full businessEntity element behave as though all categoryBag elements found at the businessEntity level and in all contained or referenced businessService elements and bindingTemplate elements are combined.

For a find_service function, specifies that the categoryBag entries for the full businessService element behave as though all categoryBag elements found at the businessService level and in all contained or referenced elements in the bindingTemplate elements are combined.

This value applies only to the find_business and find_service functions.

diacriticInsensitiveMatch
Specifies that the matching behavior for name, keyValue, and keyName, where applicable, is performed without regard to diacritics, for example accent marks. This is an optional value that applies to the find_business, find_service, find_binding, find_tModel, and find_relatedBusinesses functions.

diacriticSensitiveMatch
Specifies that the matching behavior for name, keyValue, and keyName, where applicable, is performed with regard to diacritics, for example accent marks. This is the default behavior. This value applies to the find_business, find_service, find_binding, find_tModel, and find_relatedBusinesses functions.

exactMatch
Specifies that only entries with names, keyValues, and keyNames, where applicable, that exactly match the name argument passed in, after normalization, are returned. The matching behavior is sensitive to case and diacritics, where applicable, and is the default behavior. This value applies to the find_business, find_service, find_binding, find_tModel, and find_relatedBusinesses functions.

signaturePresent
Specifies that the result set is restricted to either entities that contain an XML digital signature element, or entities that are contained in an entity that contains an XML digital signature element. This value applies to the find_business, find_service, find_binding, find_tModel, and find_relatedBusinesses functions.

orAllKeys
Specifies that the tModelBag and categoryBag elements use OR behavior with the keys in a bag, rather than AND behavior. It is not possible to use OR behavior with the categories and retain the default AND behavior of the tModel entities. For the find_business function, this is the default behavior for the identifierBag element. This value applies to the find_service, find_binding (for categoryBag and tModelBag) and find_tModel functions, where it is the default behavior for the identifierBag element and applies to the categoryBag element.

orLikeKeys
Specifies that when a categoryBag or identifierBag element contains multiple keyedReference elements, the elements use OR behavior with any keyedReference filters that come from the same namespace, that is, the filters have the same tModelKey value, rather than AND behavior. This value applies to the find_business, find_service, find_binding, and find_tModel functions.

serviceSubset
Specifies that the component of the search that involves categorization uses only the categoryBag elements from contained or referenced businessService elements in the registered data, and ignores any entries found in the categoryBag that are not direct descendent elements of registered businessEntity elements. This value applies only to the find_business function with the categoryBag element.

sortByNameAsc
Specifies that the result set that a find or get inquiry API returns is sorted on the name field in ascending order. This value takes precedence over sortByDateAsc and sortByDateDesc values, but if a sortByDate*Xxx* value is used without a sortByName*Xxx* value, the result set is sorted by date, regardless of the name field. This value applies to the find_business, find_service, find_tModel, and find_relatedBusinesses functions.

sortByNameDesc
Specifies that the result set that a find or get inquiry API returns is sorted on the name field in descending order. This value takes precedence over sortByDateAsc and sortByDateDesc values, but if a sortByDate*Xxx* value is used without a sortByName*Xxx* value, the result set is sorted by date, regardless of the name field. This value applies to the find_business, find_service, find_tModel, and find_relatedBusinesses functions.

sortByDateAsc
Specifies that the result set that a find or get inquiry API returns is sorted on the most recent date when each entity, or any entities that they contain, were last updated, in ascending chronological order, that is, the oldest entity is returned first. If this value is used with a sortByName*Xxx* value, the name-based sort takes precedence over the date-based sort, that is, the results are sorted by name, then within names by date, oldest to newest. This is the default behavior for the find_binding function. This value applies to the find_business, find_service, find_tModel, and find_relatedBusinesses functions.

sortByDateDesc
Specifies that the result set that a find or get inquiry API returns is sorted on the most recent date when each entity, or any entities that they contain, were last updated, in descending chronological order, that is, the most recently changed entity is returned first. If this value is used with a sortByName*Xxx* value, the name-based sort takes precedence over the date-based sort, that is, the results are sorted by name, then within names by date, newest to oldest. This value applies to the find_business, find_service, find_binding, find_tModel and find_relatedBusinesses functions.

suppressProjectedServices
Specifies that a find_service or find_business function must not return service projections. This value is enabled by default whenever the find_service function is used without a businessKey key. This value applies to the find_business and find_service functions.

For further details on the findQualifiers, refer to the UDDI Version 3 Specification documentation.

Publish API for the UDDI Version 3 registry

Use the UDDI Publish API to publish, delete, and update information that is contained in a UDDI registry. The messages that are defined in this section all behave synchronously.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java. To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Publish API calls:

add_publisherAssertions
: Adds one or more publisherAssertions to the assertion collection of an individual publisher.

delete_binding
: Deletes one or more instances of bindingTemplate data from the UDDI registry.

delete_business
: Removes one or more business registrations and all direct contents from a UDDI registry.

delete_publisherAssertions
: Removes one or more publisherAssertion elements from an assertion collection of a publisher.

delete_service
: Removes one or more businessService elements from the UDDI registry and from its containing businessEntity parent.

delete_tModel
: Logically deletes one or more tModel structures. Logical deletion hides the deleted tModel entities from find_tModel result sets, but does not physically delete them, so they are returned on a get_registeredInfo request.

get_assertionStatusReport
: Provides administrative support to determine the status of current and outstanding publisher assertions that involve any of the business registrations that the individual publisher account manages. A publisher can use this message to see the status of assertions that they have made, and to see assertions that others have made that involve businessEntity structures that the calling publisher account controls.

get_publisherAssertions
: Obtains the full set of publisher assertions that are associated with an individual publisher account. Publisher assertions are used to control publicly visible business relationships.

get_registeredInfo
: Obtains an abbreviated list of all businessEntity and tModel data that are controlled by the individual that is associated with the credentials that are passed.

save_binding
: Saves or updates a complete bindingTemplate element. This message can be used to add or update one or more bindingTemplate elements as well as the container or contained relationship that each bindingTemplate has with one or more existing businessService elements.

save_business
: Saves or updates information about a complete businessEntity element. This API has the broadest scope of all the save_*xx* API calls in the publisher API, and can be used to make comprehensive changes to the published information for one or more businessEntity elements that an individual controls.

save_service
: Adds or updates one or more businessService elements that a specified businessEntity exposes.

save_tModel
: Adds or updates one or more registered tModel entities.

set_publisherAssertions
: Manages all the tracked relationship assertions that are associated with an individual publisher account.

For full details of the Publish API syntax, refer to the UDDI Version 3 API specification.

Custody and Ownership Transfer API for the UDDI Version 3 registry

Use the UDDI Custody and Ownership Transfer API to transfer custody or ownership of one or more entities that are contained in a UDDI Version 3 registry. The UDDI Version 3 registry supports only intra-node ownership transfer; it does not support inter-node custody transfer.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java. To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Custody and Ownership Transfer API calls:

discard_transferToken
: Discards a transferToken that is obtained through the get_transferToken API.

get_transferToken
: Initiates the transfer of ownership of one or more businessEntity or tModel entities from one publisher to another. When the API is invoked, no transfer takes place. Instead, the relinquishing publisher uses this API to obtain permission from the custodial node, in the form of a transferToken, to undertake the transfer. The relinquishing publisher gives the transferToken to the recipient publisher, who must invoke the transfer_entities API to transfer the entities.

transfer_entities
: Performs the transfer of entities when called by the recipient publisher. The recipient publisher must specify an unexpired transferToken on the call.

For details of the query syntax, refer to the UDDI Version 3 API specification.

Security API for the UDDI Version 3 registry

The UDDI Version 3 registry has an independent Security API, unlike UDDI Version 1 and Version 2, where the Security API is part of the Publish API.

To access all API calls and arguments that are supported by the UDDI Version 3 registry programmatically, use the UDDI Version 3 Client for Java. To access the API functions graphically, you can use the UDDI user interface, but not all functions are available with this method.

The UDDI Version 3 registry supports the following Security API calls:

discard_authToken
: Informs a node that a previously obtained authentication token is no longer required and is no longer valid if it is used after this message is received. The token is discarded and the session is effectively ended.

get_authToken
: Requests an authentication token in the form of an authInfo element from a UDDI node.

For full details of the Security API syntax, refer to the UDDI Version 3 API specification.

UDDI registry Version 3 entity keys

The UDDI Version 3 specification expands the space available for keys. Entity keys can be any Universal Resource Identifier (URI) that follows the recommended UDDI scheme. Depending on registry policy, both the UDDI registry and the publisher of the entity can assign keys.

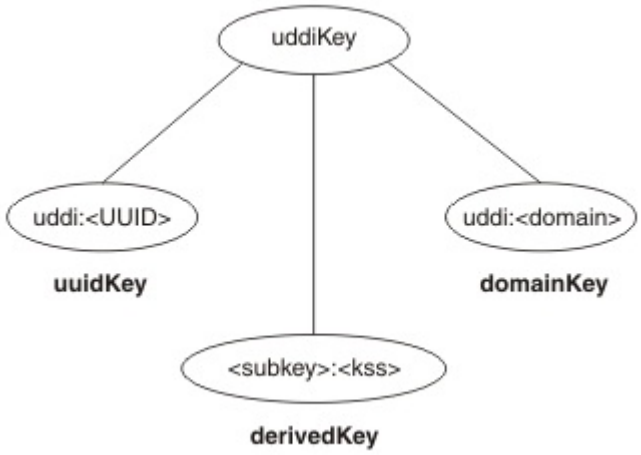
Entity keys are identifiers that are used to address entities in a UDDI registry. Each businessEntity, businessService, bindingTemplate, or tModel entity has a unique identifier that is generated or assigned when it is first published in the UDDI registry. In a particular registry, a key must be unique. For UDDI Version 1 and Version 2, the space is limited to a universal unique identifier (UUID). For UDDI Version 3, entity keys can be any URI that follows the recommended UDDI scheme.

For UDDI Version 3, depending on registry policy, not only can the UDDI registry assign keys, but also the publisher of the entity. These differences raise issues for maintaining key uniqueness and managing key space.

UDDI Scheme

The UDDI Version 3 registry implements the recommended UDDI scheme, as detailed in Section 4.4 of the UDDI Version 3 Specification. (http://uddi.org/pubs/uddi_v3.htm). This scheme defines the format of the keys, the valid characters, and the concept of key space.

In the UDDI Version 3 registry, a key is any URI and is limited to 255 characters. The following diagram shows the different types of keys in the UDDI key scheme:



All keys are composed of a set of tokens that are separated by colons (:). The first token for all keys that follow the UDDI scheme is uddi. There are three types of keys:

- The uuidKey keys contain two tokens, the mandatory uddi and a <UUID>. These keys ensure uniqueness through the UUID algorithm.
- The domainKey keys contain two tokens, the mandatory uddi and a domain name. These keys are for creating additional mutually exclusive key spaces.
- The derivedKey keys are composite keys that are based on a subkey, which is any uddiKey, and an additional token, kss, which is a key-specific string that differentiates keys. A publisher can assign the kss token, or the kss token can be calculated algorithmically (UUID).

Another concept that is included in the UDDI key scheme is a *key generator*. A key generator represents a key space. A publisher can save entities by using keys from a certain key space only if that publisher owns the key generator that represents the key space. This feature helps to secure unique keys. The key generator is a tModel entity, with a key that is in the form <subkey>:keygenerator. By owning this tModel entity, a publisher can assign keys in the form <subkey>:<kss>. The publisher can also publish new key generator tModel entities in the form <subkey>:<kss>:keygenerator.

Key uniqueness and registry root key space

You can configure instances of the UDDI registry as a root registry or as an affiliate registry.

Root registries define their own root key space by defining their own root key generator. This action defines the total key space that the registry manages. All keys that the registry generates are in this key space. If the policy allows, publishers can request subdivisions of this key space by publishing new key generator tModel entities in the form <rootkeygenerator>:<subdivisionIdentifier>:keygenerator. Publishers can then include publisher-supplied keys in subsequent publish requests that are in their allocated key space subdivision, <rootkeygenerator>:<subdivisionIdentifier>:<kss>.

To avoid key collisions, affiliate registries must establish their root key generator by first submitting a tModel:keygenerator request to the root registry they want to be an affiliate of, and then using this

subdivision of the key space of the root registry as their own root key generator. This process ensures that there are no collisions between keys that are generated or accepted by an affiliate registry, and other keys in the root registry key space.

To maintain key uniqueness, simple rules are applied. The registry generates new keys only in the key space that is defined by its own root key generator. The registry only accepts publisher-supplied keys that are in subdivisions of key space that the publisher owns, that is, as the result of a previous successful tModel tModel:keygenerator publish request.

Simple example for a private root registry:

```
with a Root keygenerator:

uddi:aPrivateRegistryKeySpaceIdentifier:keygenerator

generates Entity Keys of format:

uddi:aPrivateRegistryKeySpaceIdentifier:<uuid>

depending on Policy, accepts tModel:keygenerator requests from Publishers for
'top-level' subdivisions of format:

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:
keygenerator
```

Publishing tModel:keygenerator requests for subdivisions of key space

Depending on policy, a publisher can submit a request for a top-level subdivision of the key space in the root registry for its own use. The policy can specify whether the registry supports publisher-supplied keys and whether the user entitlements of particular publisher allow the publisher to submit requests for key space.

As well as top-level subdivisions of the key space in the root registry, a publisher can also create additional subdivisions of key space.

The following simple example is a continuation of the previous example and requests the subdivision `a`:

```
uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:
a:keygenerator
```

For a request for a further subdivision to be successful, the publisher that requested, and owns, the tModel entity for the previous subdivision must make the request. In the previous example, this publisher is uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:keygenerator).

Publishing with a publisher-supplied key

After a publisher successfully requests a subdivision of the key space in a root registry, that publisher must establish and maintain its own scheme to ensure that the keys that are generated for use as publisher-supplied keys in subsequent publish requests are unique in the subdivision.

Valid schemes need to generate keys that are unique derived keys in the allocated key space subdivision, for example by including a unique (incremented) numeric index.

The following simple example continues from the previous example. For a key space subdivision that results from the tModel:keygenerator request:

```
uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:
a:keygenerator

valid keys are:

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:1

uddi:aPrivateRegistryKeySpaceIdentifier:aPublisherSubdivisionIdentifier:a:2
```

Digital signatures and the UDDI registry

In UDDI Version 3, publishers can digitally sign UDDI elements while they are publishing. The UDDI Version 3 schema supports the signing of businessEntity, businessServices, bindingTemplate, tModel, and publisherAssertion elements.

You can validate UDDI elements that are digitally signed to prove that they have not been modified or tampered with, and that their integrity is intact.

For full details about signing UDDI entities and verifying signatures, see *Appendix I: Support for XML Digital Signatures* in the UDDI Version 3.0.2. specification.

The UDDI registry does not validate signatures when signed elements are published. When the signed elements are retrieved, the retrieving client is responsible for validating the signature and providing a mechanism to ensure that the signer certificate is signed by a Certificate Authority (CA) that the client approves and trusts. If a signature is decrypted successfully by using the signer public key, it indicates that only the owner of the corresponding private key can have signed and published this element.

Signature generation

The attributes of an element are included in the generation of an element signature. Therefore, all entity keys must be available when the signature is generated. Publishers can generate publisher-assigned keys for all the keys of an element before signing. Alternatively, if publishers publish the element without keys, the registry node generates the required entity keys and then retrieves, signs, and republishes the signed element.

Signature validation

The signature element to validate is in the top-level element that a call to the getXXDetails method returns. The client is responsible for the validation. The client must have previously imported the X509.3 certificate of the publisher, and validated that certificate based on the CA it trusts. In this way, the client has access to the public validation key of the publisher that corresponds to the private signing key that the publisher used to sign the entity before publishing it.

You can use the UDDI Version 3 Client to construct Java API for XML-based RPC (JAX-RPC) objects and to invoke the UDDI Version 3 web service. As part of this client, you can use a helper class, com.ibm.uddi.v3.client.apilayer.xmldig.SignatureUtilities, to create and validate digital signatures on the UDDI Version 3 entities that support them. For details of application programming interfaces (APIs) in this helper class and the SignatureUtilitiesException exception, see the API information.

For UDDI, digital signatures are used to sign the data. They are not used to authenticate the SOAP message.

UDDI Version 3 Client

You can use the UDDI Version 3 Client for Java to access the UDDI Version 3 application programming interfaces (APIs) from Java client code.

The UDDI Version 3 Client for Java is a Java API for XML-based RPC (JAX-RPC) class library that provides an API that client programs can use to interact with a Version 3 UDDI registry. This class library can be used to construct UDDI JAX-RPC objects and to invoke the UDDI Version 3 web service.

This client also contains an XML digital signature utility class called SignatureUtilities, which is provided to construct and validate digital signatures on UDDI elements.

Multiple language encoding support

The UDDI Version 3 API supports both UTF-8 and UTF-16 encoding. Internally, UTF-16 characters are stored as UTF-8 characters. This behavior is not apparent to the user application.

Client JAR file

WebSphere Application Server provides the class library `uddiv3client.jar`, which contains the JAX-RPC UDDI Version 3 types and UDDI WebService invocation classes. This Java archive (JAR) file is in *app_server_root*/UDDIReg/clients.

The UDDI Version 3 client provides port types that map onto the UDDI Version 3 SOAP Inquiry, Publish, Custody Transfer, and Security APIs. These APIs are protected by access control mechanisms, for example role mappings. A client program that uses the UDDI Version 3 client gets the appropriate port type for the request that is issued, for example, the UDDI_Publication_PortType for a save_business request. If the role mappings are such that the request requires a WebSphere Application Server authenticated user ID, the client program passes the user ID and password by setting the relevant properties on the JAX-RPC stub for that port.

UDDI Version 3 Client samples

The following UDDI registry samples illustrate the use of the Version 3 Client. To access the samples, see the Samples for WebSphere Application Server and use the UDDI Registry link.

UDDIv3ClientBindingSample.java
An example of how to save and find binding templates.

UDDIv3ClientBusinessSample.java
An example of how to save and find business entities.

UDDIv3ClientServiceSample.java
An example of how to save and find business services.

UDDIv3ClientSignedBusinessSample.java
An example of how to sign and verify a business entity.

UDDIv3ClientTModelSample.java
An example of how to save and find TModel entities.

UDDIv3ClientSignedTModelSample.java
An example of how to sign and verify TModel entities.

These classes contain details on how to compile and run the samples.

HTTP GET services for UDDI registry data structures

The UDDI registry offers an HTTP GET service for access to the XML representations of the businessEntity, businessService, bindingTemplate, and tModel UDDI data structures. The Uniform Resource Locators (URLs) at which these structures are accessible use the entity key as a URL parameter. The XML element that is returned is a businessDetail, serviceDetail, bindingDetail or tModelDetail element, according to the type of entity key that is supplied.

XML for both UDDI Version 2 and Version 3 can be retrieved, each at different URLs. The URLs to send the HTTP GET requests to are in the following formats:

For UDDI Version 2:
http://*server*:*port*/uddisoap/get?*entityKey_type=v2_entityKey*

For UDDI Version 3:
http://*server*:*port*/uddiv3soap/get?*entityKey_type=v3 entityKey*

For example, if the server is myserver.com and the port is 9080, you can access the uddi-org:types tModel entity at the following URLs:

UDDI Version 2:
http://myserver.com:9080/uddisoap/get?tModelKey=uuid:c1acf26d-9672-4404-9d70-39b756e62ab4

UDDI Version 3:
http://myserver.com:9080/uddiv3soap/get?tModelKey=uddi:uddi.org:categorization:types

A number of UDDI node property and policy settings relate to the HTTP GET services:

- Version 3 HTTP GET for UDDI entities
  - Node supports HTTP GET
  - URL prefix for Version 3 GET servlet
  - Node generates discovery URLs
- Version 2 HTTP GET for discovery URLs
  - Prefix for generated discovery URLs
  - Node generates discovery URLs

UDDI registry SOAP service end points

UDDI Version 3 supports multiple versions of SOAP API services. Depending on the security settings for WebSphere Application Server and the user data constraint transport guarantee settings for the UDDI SOAP service, UDDI Version 3 supports different end points for different services.

The following list of default context root and Uniform Resource Locator (URL) values apply when WebSphere Application Server security is enabled and you and do not change the default supplied security settings. If you do not use the default security settings, the context root and URL values might differ.

In the following URLs, the variables have the following values:

- *host_name* is the name of the machine that is running the relevant profile.
- *http_port* is the internal HTTP port for the profile, for example 9080.
- *ssl_port* is the internal secure sockets layer (SSL) port for the profile, for example 9443.
- Version 1 and Version 2 SOAP API services

  Inquiry service
  Default (soap.war) context-root='/uddisoap' and url-pattern = 'inquiryAPI' or 'inquiryapi'.

  Default URL: http://*host_name*:*http_port*/uddisoap/inquiryapi

  Publish service
  Default (soap.war) context-root='/uddisoap' and url-pattern = 'publishAPI' or 'publishapi'.

  Default URL: https://*host_name*:*ssl_port*/uddisoap/publishapi or http://*host_name*:*http_port*/uddisoap/publishapi
- Version 3 SOAP API services

  Inquiry service
  Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Inquiry_Port'

  Default URL: http://*host_name*:*http_port*/uddiv3soap/services/UDDI_Inquiry_Port

  Publish service
  Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Publish_Port'

  Default URL: https://*host_name*:*ssl_port*/uddiv3soap/services/UDDI_Publish_Port or http://*host_name*:*http_port*/uddiv3soap/services/UDDI_Publish_Port

  Custody transfer service
  Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Custody_Port'

Default URL: https://hostname:9443/uddiv3soap/services/UDDI_Custody_Port or
http://hostname:9080/uddiv3soap/services/UDDI_Custody_Port

Security service

Default (soap.war) context-root='/uddiv3soap' and url-pattern = '/services/UDDI_Security_Port'

Default URL: https://*host_name*:*ssl_port*/uddiv3soap/services/UDDI_Security_Port or
http://*host_name*:*http_port*/uddiv3soap/services/UDDI_Security_Port

An endpoint is available for using HTTP GET to return XML representations of UDDI entities. See the information about HTTP GET services for UDDI registry data structures.

If you configure the UDDI registry to use WebSphere Application Server security, and you do not change the default data confidentiality settings for the UDDI SOAP service, services with default end point URLs with HTTPS and a SSL port require that their data is transported confidentially. Requests that do not use HTTPS are rejected.

If you configure the UDDI registry to use WebSphere Application Server security and you change the data confidentiality setting for the UDDI SOAP service to NONE, or you disable WebSphere Application Server security, services with default end point URLs with HTTPS and a SSL port can also use HTTP and a HTTP port.

To understand how access to the SOAP APIs is protected, see the information about access control for UDDI registry interfaces.

UDDI4J programming interface (Deprecated)

The UDDI4J Version 2 APIs are deprecated in this version of WebSphere Application Server. The UDDI Version 3 Client for Java is the preferred API for accessing UDDI through Java code.

WebSphere Application Server provides UDDI4J classes in the com.ibm.uddi.jar file. This file contains classes that support Version 1 and Version 2 of the UDDI specification, providing compatability with earlier versions of WebSphere Application Server. The UDDI4J classes in this file are deprecated.

The UDDI4J methods map onto the UDDI Version 1 and Version 2 SOAP Inquiry and Publish APIs. These APIs are protected by access control mechanisms, for example role mappings. If the role mappings for these APIs are such that requests to these interfaces require a WebSphere Application Server authenticated user ID, a client program that uses UDDI4J must pass the user name and password, by setting the system properties http.basicAuthUserName and http.basicAuthPassword. A UDDI4J client program can also specify details for a proxy server, including a user name and password, by using the following system properties:

- http.proxyHost
- http.proxyPort
- http.proxyUserName
- http.proxyPassword

Using the UDDI EJB Interface (Deprecated)

Use the Enterprise JavaBeans (EJB) application programming interface (API) of the UDDI registry component to publish, find, and delete UDDI entries. However, the UDDI EJB interface is deprecated and supports UDDI version 2 API requests only.

Before you begin

Both WebSphere Application Server and the UDDI registry must be installed, and must both be running. You cannot use the EJB client from a machine that does not have WebSphere Application Server installed.

About this task

Note: The UDDI EJB interface is deprecated in WebSphere Application Server Version 6.0.

The client classes that are required for the EJB interface are contained in *app_server_root*/UDDIReg/clients/uddiejbclient.jar. For the Java documentation for these classes, see the information about additional APIs.

The EJB API is contained in two stateless session beans, one for the inquiry API (com.ibm.uddi.ejb.InquiryBean) and one for the publish API (com.ibm.uddi.ejb.PublishBean), whose public methods form an EJB interface for the UDDI registry. All the public methods on the InquiryBean class correspond to UDDI Version 2 inquiry API functions, and all the public methods on the PublishBean class correspond to UDDI Version 2 publish API functions. Not all UDDI Version 2 API functions are implemented, for example get_authToken, discard_authToken, and get_businessDetailExt.

In each interface, there are groups of overloaded methods that correspond to the operations in the UDDI 2.0 specification. There is a separate method for each major variation in function. For example, the single UDDI operation find_business is represented by ten variations of findBusiness methods, with different variations to find by arguments such as name or categoryBag.

The arguments for the EJB interface methods are Java objects in the com.ibm.uddi.datatypes package. Generally, there is a one-to-one correspondence between classes in this package and elements of the UDDI Version 2 XML schema. There are exceptions to this correspondence, for example, where UDDI XML elements can be represented by a single string. For more information, see the Java documentation for the package com.ibm.uddi.datatypes in the information about additional APIs.

The methods on the EJB InquiryBean class map to the EJB inquiry role, and those of the EJB PublishBean class map to the EJB publish role. The EJB inquiry and publish roles protect the EJB interface, as described in information about access control for UDDI registry interfaces. If the role mapping is such that a method requires a WebSphere Application Server authenticated user ID, a client program can supply the user ID and password, either when prompted by WebSphere Application Server, or by providing application code that logs in to the default realm using the user ID and password. Use the `sas.client.props` configuration file to determine how to specify the user ID and password when you configure security.

To use the EJB client, use the following steps.

Procedure

1. Set up your environment to communicate with WebSphere Application Server:

```
. app_server_root/bin/setupCmdLine.sh
```

   Notice that a single space character follows the period (.)

2. Ensure that your CLASSPATH includes the uddiejbclient.jar file (from the *app_server_root*/UDDIReg/clients directory), and the code for your client.
3. Compile your EJB client programs:

```
$JAVA_HOME/bin/javac -extdirs $WAS_EXT_DIRS:$JAVA_HOME/jre/lib/ext
-classpath $WAS_CLASSPATH:$CLASSPATH yourcode.java
```

4. Run the compiled programs:

```
$JAVA_HOME/bin/java -Djava.ext.dirs=$WAS_EXT_DIRS:$JAVA_HOME/jre/lib/ext
-Dwas.install.root=$WAS_HOME -Dserver.root=$WAS_HOME $CLIENTSAS $CLIENTSOAP
-cp $WAS_CLASSPATH:$WAS_HOME/UDDIReg/clients/uddiejbclient.jar:$CLASSPATH
<class name> <args>
```

   Ensure that your PATH statement starts with *app_server_root*/java/bin.

Using the UDDI registry user interface

The UDDI registry user interface (also referred to as the UDDI registry user console) is a graphical interface that you can use to issue inquiry requests and explore the UDDI registry.

Before you begin

If you require multiple language encoding, the UDDI user console supports only Universal Transformation Format (UTF)-8 encoding. For UTF-8 encoding support, configure the application server that hosts the UDDI registry for UCS transformation format. The UDDI user console does not support UTF-16 encoding.

About this task

The UDDI user console provides a graphical user interface to most of the UDDI Version 3 API. The user console does not support the full API set, but provides a way to issue inquiry requests and to familiarize yourself with general UDDI concepts. Explanatory text on the screens provides help. The following list describes areas that are not supported through the user console, and other known restrictions in the user console.

- You cannot specify maximum rows on find queries. You can set the “Single maximum rows” value for the registry by using the “Maximum inquiry result set size” general property on the administrative console.
- The identifier feature is not supported in the find business area or the find technical model (tModel) area.
- In the add business area, adding discovery URLs, identifiers or digital signatures is not supported.
- In the add technical model (tModel) area, adding identifiers or digital signatures is not supported.

The exact behavior of the user console depends on the following configurable factors:

- Whether WebSphere Application Server security is enabled.
- How the UDDI registry GUI role mappings are set. The UDDI registry supports two security roles for the user console:
  - GUI_Publish_User
  - GUI_Inquiry_User.
- How the UDDI registry GUI secure sockets layer (SSL) transport guarantee constraints are set. The UDDI registry supports the configuration of SSL settings, including two settings for the user console.

The following table summarizes the behavior of the UDDI registry user console.

Table 246. Behavior of the UDDI user console. The table lists the URLs used for the different UDDI registry settings, along with a description of the behavior associated with each setting.

WebSphere Application Server security status	URL used to access the UDDI user console	Behavior of the UDDI user console
Enabled	http://*host_name*:*http_port*/uddigui	Inquiry requests do not require authentication; they use the HTTP URL and are not secure. Publish requests require WebSphere Application Server authentication. When you access the publish pane, you are dynamically redirected to use HTTPS, and are prompted for a user ID and password. For a successful request, the authenticated user must be registered as a UDDI publisher. If the GUI_Inquiry_User role is mapped to all authenticated users, and the transport guarantee in the user data constraint section for that role is set to CONFIDENTIAL, all requests, including inquiry, require authentication and use of HTTPS.
	https://*host_name*:*ssl_port*/uddigui	Requests are secure; you are prompted to authenticate with a user ID and password. For a successful request, the authenticated user must be registered as a UDDI publisher.

Table 246. Behavior of the UDDI user console (continued). The table lists the URLs used for the different UDDI registry settings, along with a description of the behavior associated with each setting.

WebSphere Application Server security status	URL used to access the UDDI user console	Behavior of the UDDI user console
Disabled	http://*host_name*:*http_port*/uddigui	No requests, either publish or inquire, are authenticated and the data flow is not secure (non- SSL). Even though SSL transport-guarantee settings are defined, they are not enforced if security is disabled. All publish operations are performed using a user ID of UNAUTHENTICATED, or a value that can be configured by using the administrative console or the Java Management Extensions (JMX) management interface (this applies to new requests only).
	https://*host_name*:*ssl_port*/uddigui	No requests, either publish or inquire, are authenticated, but the data flow is secure because the SSL URL and port are used explicitly. All publish operations are performed using a user ID of UNAUTHENTICATED, or a value that can be configured by using the administrative console or the JMX management interface (this applies to new requests only).

The variables in the table have the following values:

- *host_name* is the name of the machine that is running the relevant profile.
- *http_port* is the internal HTTP port for the profile, for example 9080.
- *ssl_port* is the internal SSL port for the profile, for example 9443.

To display the UDDI registry user console, use the following procedure.

Procedure

1. Start the UDDI application, if it is not already running.
2. Open a browser window and ensure that cookies are enabled.
3. Access the UDDI registry user console through one of the following default URLs.
   - http://*host_name*:*http_port*/uddigui
   - https://*host_name*:*ssl_port*/uddigui
4. Optional: To change the appearance and operation of the UDDI registry user console, modify the appropriate `.css` stylesheet files. You can edit style class definitions in the stylesheet files, including font attributes, layout, and colors. The stylesheet files are in the following directory:

```
profile_root/installedApps/cell_name/
UDDIRegistry.node_name.server_name.ear/v3gui.war/theme
```

   After you change a stylesheet file, refresh the browser window for the changes to take effect.

Results

The user console displays the default frameset, which contains the following items:

- The header frame.
- The navigation frame, which shows find options.
- The details frame.

What to do next

You can now use the UDDI user console to find, edit, or publish UDDI information.

Finding an entity by using the UDDI registry user interface

You can use the UDDI registry user interface to find services, businesses, and technical models.

Before you begin

Ensure that the UDDI registry application is started and the UDDI registry user console is displayed.

About this task

To use the UDDI registry user interface (also referred to as the UDDI registry user console) to find services, businesses, and technical models, use the following procedure.

Procedure

1. In the UDDI registry console, activate the Find pane. Either click the **Find** tab, or click the **Find** link at the top of the page or on the Welcome page.
2. Optional: For a simple search without any find qualifiers, use the following steps:
   a. In the **Quick Find** section of the **Find** tab, select the kind of entity that you want to find; service, business, or technical model.
   b. In the **Starting with** field, enter the name of the entity. Use the percent (%) wildcard character to search for a partial name.
   c. Click **Find**.

   The results are displayed.
3. Optional: For advanced search, use the following steps:
   a. In the **Advanced Find** section of the **Find** tab, click the appropriate link for the kind of entity that you want to find; service, business, or technical model. The advanced search form is displayed in the frame.
   b. Enter your search criteria in the advanced search form, and select any find qualifiers that you require.

      You must enter at least one name to search for, by using the **Add Name** link. You can use this link to enter multiple names.

      You can also add multiple categorizations. To add a categorization, use the **Show category tree** link in the **Categorizations** section to display, a tree of categories (or taxonomies) that define the types of item to find according to various classification systems. Expand the tree to find the category that you want, click the category to add the information to the advanced search form, then use the **Add Categorization** link to include the category in the search.
   c. Click **Find *entities***.

   The results are displayed in the detail frame.

Publishing an entity by using the UDDI registry user interface

You can use the UDDI registry user interface to publish businesses and technical models.

Before you begin

Ensure that the UDDI registry application is started and the UDDI registry user console is displayed.

About this task

You can use the UDDI registry user interface (also referred to as the UDDI registry user console) to publish businesses and technical models.

To publish a service, publish a business, and then edit the entity to add a service to that business.

Procedure

1. In the UDDI registry console, activate the Publish pane. Either click the **Publish** tab, or click the **Publish** link at the top of the page or on the Welcome page.
2. Optional: To publish an entity by name only, use the Quick publish section and the following steps:
   a. In the **Quick Publish** section of the **Publish** tab, select the kind of entity that you want to publish; business or technical model.

b. In the **Name** field, enter the name of the entity.
c. Click **Publish**.

The details of the published entity are displayed.

3. Optional: To publish an entity with more information, use the following steps:
   a. In the **Advanced Publish** section of the **Publish** tab, click the appropriate link for the kind of entity you want to publish; business or technical model. The advanced publish form is displayed.
   a. Enter the details for the entity in the advanced publish form. You can enter multiple names, descriptions, contacts, or categorizations by using the relevant **Add** link.

      To add a categorization, use the **Show category tree** link in the **Categorizations** section to display a tree of categories (or taxonomies) that define the types of item to publish according to various classification systems. Expand the tree to find the category that you want, click the category to add the information to the advanced publish form, then click the **Add Categorization** link.
   b. Click **Publish *entity*** to publish the business or technical model to the UDDI registry.

Editing or deleting an entity by using the UDDI registry user interface

You can use the UDDI registry user interface to edit or delete the businesses and technical models that you own, for example, to add services to businesses.

Before you begin

Ensure that the UDDI registry application is started and the UDDI registry user console is displayed.

About this task

To use the UDDI registry user interface (also referred to as the UDDI registry user console) to edit or delete the businesses and technical models that you own, or add services to businesses, use the following steps.

Procedure

1. In the UDDI registry console, activate the Publish pane. Either click the **Publish** tab, or click the **Publish** link at the top of the page or on the Welcome page.
2. In the Registered Information section on the **Publish** tab, click **Show owned entities** to show the businesses and technical models that you registered in the UDDI registry.
3. Optional: To delete a business or a technical model, click **Delete** in the **Actions** column for that entity. When you delete a technical model, it is hidden, rather than physically deleted, as specified by the UDDI Version 3.0 specification. If you click **Shown owned entities**, the technical model is still displayed, but if you use the Find function, the technical model is not displayed. All other entities are deleted from the UDDI registry in the usual way.
4. Optional: To edit a business or technical model, click **Edit** in the **Actions** column for that entity, enter the required details, then click **Update *entity*** to save the changes in the UDDI registry.
5. Optional: To add a service to a business, click **Add service** in the **Actions** column for that business. Enter the details, then click **Add Service** to publish the service to the UDDI registry. The service details are displayed.

Creating business relationships by using the UDDI registry user interface

If your business has an association with another business in the UDDI registry, for example a preferred supplier, you can describe this association in the UDDI registry by creating a *business relationship*.

Before you begin

- The UDDI registry contains the following default relationship types:

Parent-child
A hierarchical relationship exists between the two business entities, which might represent, for example, a large organization and a subsidiary.

Peer-peer
The two business entities represent peer organizations, for example a company and its supplier.

Identity
The two business entities represent the same organization.

If you require a different relationship type, create a user-defined value set to represent the relationship type that you require, as described in the topic about user-defined value set support in the UDDI registry.

- Each business that is involved in the relationship must already exist in the UDDI registry.
- Ensure that the UDDI registry application is started and the UDDI registry user console is displayed.

About this task

Complete this task when you want to publicize an association between two businesses in the UDDI registry. For example, your organization, represented by a business entity in the UDDI registry, might have several departments, each one represented by a different business entity in the UDDI registry. You might want to declare these departments as being linked to the parent organization, by creating parent-child relationships between the appropriate business entities.

Procedure

1. Activate the **Publish** tab by clicking it, or by clicking the **Publish** link at the top of the page or on the Welcome page.
2. Under **Registered Information**, click **Show owned entities**. The entities that you own are displayed in the detail frame on the right.
3. In the section for the businesses that you own, find the business that you want to link from, and click the **Add relationship** link for that business. The Add Business Relationship pane is displayed. The business key for the business that you selected is already listed in the From section.
4. Click **Add** to add the second business, the business that you want to link to. The advanced find pane is displayed.
5. Find the second business, as described in the advanced find step of “Finding an entity by using the UDDI registry user interface” on page 1793.
6. Click **Select** to add the second business to the relationship. If you want to change the positions of the businesses, click **Swap**.
7. Select the type of relationship from the **Type** list.
8. If required, type a description in the **Usage** field.
9. Click **Add relationship** to create the relationship. If you own both businesses, no further action is required. The relationship is displayed as a *publisher assertion* in the list of entities that you own. The status of the assertion is complete.
10. If you do not own the second business, the status of the assertion is pending. The owner of the second business must create a relationship from their business to yours, for the relationship to be complete and visible to other parties. The relationship type must match the type that you chose earlier.

Results

The business relationship is published to the UDDI registry. The UDDI user interface only shows publisher assertions for entities that you own. To view other relationships, use the UDDI inquiry API provided.

What to do next

For more information about publisher assertions, refer to the UDDI specification.

To remove an assertion that you own, display your owned entities and click the **Delete** link for the relevant publisher assertion. If the business in the To field is owned by someone else, the status of the assertion becomes pending, and the relationship is no longer visible to other parties.

Example: Publishing a business, service, and technical model using the UDDI registry user interface

This example describes how to use the UDDI registry user interface to publish a used car business called Modern Cars to the UDDI registry, and how to publish a service and technical model for the business.

Before you begin, ensure that the UDDI registry application is started and the UDDI registry user console is displayed.

Adding the business

1. Click the **Publish** tab to activate the Publish pane.
2. Under **Advanced Publish**, click **Add a business**. The advanced publish form is displayed.
3. Type `Modern Cars` in the **Name** field for the business. Select the language of the business name from the list, then click **Add Name** to add the name to the business.
4. Type a description for the business, for example `Used cars for sale` in the **Description** field. Select the language of the description from the list, then click **Add Description** to add the description to the business. You can add multiple descriptions in a variety of languages as required.
5. In the **Contact** section, type your name as a contact for customers of the business. Select the language as before, and click **Add Contact**. The business contact form is displayed. Enter the details, using the **Add *entity*** links to add the information as you reach the end of each subsection. All the text fields in the form are cleared when you click an **Add *entity*** link. Click **Add Contact** to save the contact information into the Modern Cars business.
6. Use the **Categorizations** section to describe the Modern Cars business according to the NAICS 2002 categorization system:
   a. Click **Show category tree** to display the various categorization systems.
   b. Expand the **NAICS 2002** tree, then expand **Retail Trade [44]** > **Motor Vehicle and Parts Dealers [441]** > **Automobile Dealers [4411]** > **Used Car Dealers [44112]**. Click the **Used Car Dealers [441120]** category to add the category type, key name, and key value to the advanced publish form.
   c. Click **Add Categorization** to add the information to the business.
   d. Close the category tree by clicking **Close** .
7. Click **Publish Business** to publish the Modern Cars business to the UDDI registry. The details of the business are displayed.

Adding a service to the business

1. Click the **Publish** tab to activate the Publish pane.
2. Under **Registered Information**, click **Show owned entities** to display the Modern Cars business and any other entities that you own.
3. Click **Add service** in the **Actions** column of the Modern Cars business. The publish service page is displayed.
4. Add a name and description for the service, in the same way as for the business itself.
5. Click **Add a Service Binding** to display the service binding form.
   a. Enter an access point (the URL for the service on the network) and a description for the service binding.

b. Click **Add Technical Model Instance Information** to display a page where you can describe and publish a technical model instance for the service binding.
c. In the technical model information page, click **Add Technical Model**.
d. Search for the technical model that the instance uses, select the technical model from the results, then click **Add**.
e. Complete the other fields on the form and click **Add Technical Model Instance**.
f. Click **Add Binding** to save the information into the service.

Enter an access point (the URL for the service on the network) and a description for the service binding.

6. Add a categorization in the same way as for the business.
7. Click **Add Service** to publish the service to the UDDI registry.

Adding a technical model

1. In the **Advanced Publish** section, click **Add a technical model** to display the publish technical model form.
2. Add a name and description for the technical model, in the same way as for the business and the service.
3. Click **Add an Overview Document** to display the overview document form. The overview document describes the technical model.
   a. Type the location of the overview document in the **Overview URL** field.
   b. Click **Add Overview Document URL**.
   c. Add a description and click **Add Overview Document** to save the information in the technical model.
4. Add a categorization in the same way as for the business.
5. Click **Publish Technical Model** to publish the technical model to the UDDI registry.

Using the JAXR provider for UDDI

To get started with the Java API for XML Registries (JAXR) provider, you can use a sample program. You also need to consider class libraries, authentication and security, internal taxonomies, and logging and messages.

About this task

Note: From WebSphere Application Server Version 8.0, Java API for XML Registries (JAX-R) APIs are deprecated. The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery and Integration (UDDI) 2 technology, which is no longer relevant. If your applications use JAX-R, then you might consider using UDDI 3.

Procedure

- To obtain the ConnectionFactory instance, create a connection to the registry, and save an organization in the registry, see the following sample program.

```
import java.net.PasswordAuthentication;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Properties;
import java.util.Set;

import javax.xml.registry.BulkResponse;
import javax.xml.registry.BusinessLifeCycleManager;
import javax.xml.registry.Connection;
import javax.xml.registry.ConnectionFactory;
import javax.xml.registry.JAXRException;
import javax.xml.registry.RegistryService;
```

```
import javax.xml.registry.infomodel.Key;
import javax.xml.registry.infomodel.Organization;

public class JAXRSample
{
    public static void main(String[] args) throws JAXRException
    {
        //Tell the ConnectionFactory to use the JAXR provider for UDDI
        System.setProperty("javax.xml.registry.ConnectionFactoryClass",
                           "com.ibm.xml.registry.uddi.ConnectionFactoryImpl");
        ConnectionFactory connectionFactory = ConnectionFactory.newInstance();

        //Set the URLs for the UDDI inquiry and publish APIs.
        //These must be the URLs of the UDDI version 2 APIs.
        Properties props = new Properties();
        props.setProperty("javax.xml.registry.queryManagerURL",
                          "http://localhost:9080/uddisoap/inquiryapi");
        props.setProperty("javax.xml.registry.lifeCycleManagerURL",
                          "http://localhost:9080/uddisoap/publishapi");
        connectionFactory.setProperties(props);

        //Create a Connection to the UDDI registry accessible at the above URLs.
        Connection connection = connectionFactory.createConnection();

        //Set the user ID and password used to access the UDDI registry.
        PasswordAuthentication pa = new PasswordAuthentication("Publisher1",
                        new char[] { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' });
        Set credentials = new HashSet();
        credentials.add(pa);
        connection.setCredentials(credentials);

        //Get the javax.xml.registry.BusinessLifeCycleManager interface,
        //which contains methods corresponding to UDDI publish API calls.
        RegistryService registryService = connection.getRegistryService();
        BusinessLifeCycleManager lifeCycleManager =
                                  registryService.getBusinessLifeCycleManager();

        //Create an Organization (UDDI businessEntity) with name
        //"Organization 1".
        Organization org =
                         lifeCycleManager.createOrganization("Organization 1");

        //Add the Organization to a Collection, ready to be saved in the UDDI
        //registry.
        Collection orgs = new ArrayList();
        orgs.add(org);

        //Save the Organization in the UDDI registry.
        BulkResponse bulkResponse = lifeCycleManager.saveOrganizations(orgs);

        //Obtain the Organization Key (the UDDI businessEntity
        //businessKey) from the response.
        if (bulkResponse.getExceptions() == null)
        {
        //1 Organization was saved, so 1 key will be returned in the
        //response collection
            Collection responses = bulkResponse.getCollection();
            Key organizationKey = (Key)responses.iterator().next();
            System.out.println("\nOrganization Key = " +
                                                    organizationKey.getId());
        }
    }
}
```

- Ensure that the class path is set.

  The class libraries of the JAXR provider for UDDI are in the com.ibm.uddi_1.0.0.jar file, in the *app_server_root*/plugins directory. When you use the JAXR API from a Java EE application running under WebSphere Application Server, all required classes are automatically on the class path. When you use the JAXR API from outside this environment, the following .jar files must be on the Java class path:

  – *app_server_root*/lib/bootstrap.jar
  – *app_server_root*/plugins/com.ibm.uddi_1.0.0.jar
  – *app_server_root*/plugins/com.ibm.ws.runtime_6.1.0

- To use the JAXR provider for UDDI, first you must specify the name of the ConnectionFactory implementation class. Set the javax.xml.registry.ConnectionFactoryClass system property to com.ibm.xml.registry.uddi.ConnectionFactoryImpl.

  If you do not set this system property, the value defaults to com.sun.xml.registry,common.ConnectionFactoryImpl, which cannot be found and causes a JAXRException exception when the ConnectionFactory.newInstance() method is called.

  The JAXR provider for UDDI does not support lookup of the ConnectionFactory using Java Naming and Directory Interface (JNDI).
- To specify connection-specific properties, set a java.util.Properties object on the JAXR ConnectionFactory before you obtain a connection.

  The full list of these properties is in the JAXR specification. The following table lists the three most important properties, and what values they take to use the JAXR provider for UDDI to access the UDDI registry.

Table 247. Connection-specific properties required to use the JAXR provider for UDDI to access the UDDI registry. The table lists the different properties along with a description for each one.

Property	Description
javax.xml.registry.queryManagerURL	The URL for the Inquiry API of the UDDI registry for UDDI Version 2. Typically, this property is in the form: `http://hostname:port/uddisoap/inquiryapi`. This property is required.
javax.xml.registry.lifeCycleManagerURL	The URL for the Publish API of the UDDI registry for UDDI Version 2. Typically, this property is in the form: `http://hostname:port/uddisoap/publishapi`. If you do not specify this property, it defaults to the value of the javax.xml.registry.queryManagerURL property. However, the UDDI registry typically has different URLs for the Inquiry and Publish APIs, so it is advisable to specify both properties.
javax.xml.registry.authenticationMethod	The method of authentication to use when authenticating with the registry. This can take one of two values, UDDI_GET_AUTHTOKEN and HTTP_BASIC. If you do not specify a value, the default value is UDDI_GET_AUTHTOKEN. See the following step for more information.

  The only required connection property is javax.xml.registry.queryManagerURL. However, it is advisable to set javax.xml.registry.lifeCycleManagerURL and understand the default value of javax.xml.registry.security.authenticationMethod. The other connection properties that are defined in the JAXR specification are optional, and their values are not specific to the UDDI registry. The JAXR provider for UDDI does not define any additional provider-specific properties.
- To determine which method the JAXR provider uses to authenticate with the UDDI registry, set the javax.xml.registry.authenticationMethod connection property.

  The javax.xml.registry.authenticationMethod connection property determines which method the JAXR provider uses to authenticate with the UDDI registry. Two values of this property are supported:

  – UDDI_GET_AUTHTOKEN

    The JAXR provider uses the UDDI V2 get_authToken API to authenticate with the registry. The JAXR provider makes the get_authToken call automatically when the connection credentials are set. The JAXR provider saves the UDDI V2 authToken that the call returns to use on subsequent UDDI publish API calls.

  – HTTP_BASIC

    The JAXR provider uses HTTP basic authentication to authenticate with the registry. WebSphere Application Server supports HTTP basic authentication when security is enabled. The JAXR provider does not make a get_authToken call. Instead, whenever there is a UDDI API call (both Inquiry and Publish), the user name and password are sent in the HTTP headers, using HTTP basic authentication. If the UDDI registry does not require HTTP basic authentication, the credentials are ignored.

  The JAXR provider for UDDI does not support the CLIENT_CERTIFICATE or MS_PASSPORT methods of authentication.

If you do not set this property, the default authentication method is UDDI_GET_AUTHTOKEN.

- To use Secure Sockets Layer (SSL) to encrypt HTTP traffic between the JAXR provider for UDDI and the UDDI registry, see “Using SSL with the UDDI JAXR provider” on page 1802.
- To supply a custom internal taxonomy, see “Creating a custom internal taxonomy for the JAXR provider” on page 1802.
- To switch on UDDI4J logging, set the org.uddi4j.logEnabled system property to `true`. The JAXR provider for UDDI uses UDDI4J Version 2 to communicate with the UDDI registry, and UDDI4J has its own logging.

Java API for XML Registries (JAXR) provider for UDDI

The Java API for XML Registries (JAXR) is a Java client API for accessing both UDDI (Version 2 only) and ebXML registries. It is part of the Java Platform, Enterprise Edition (Java EE) specification.

Note: From WebSphere Application Server Version 8.0, Java API for XML Registries (JAX-R) APIs are deprecated. The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery and Integration (UDDI) 2 technology, which is no longer relevant. If your applications use JAX-R, then you might consider using UDDI 3.

The JAXR API comprises the Java EE packages javax.xml.registry and javax.xml.registry.infomodel. Java EE API documentation is at Web Services Reference.

The preferred UDDI Java client APIs are:

- UDDI4J Version 2, for UDDI Version 2
- UDDI Version 3 Client for Java, for UDDI Version 3

JAXR provider

The current JAXR specification (Version 1.0) defines a JAXR provider as an implementation of the JAXR API. Generally, a JAXR provider can be a JAXR provider for UDDI, a JAXR provider for ebXML, or a pluggable provider that supports both UDDI and ebXML. The JAXR provider for UDDI is a provider for UDDI only.

UDDI versions

A JAXR provider for UDDI accesses a UDDI registry that uses the UDDI Version 2 SOAP APIs only. The UDDI registry for UDDI Version 3 in this version of WebSphere Application Server supports the UDDI Version 1, 2 and 3 SOAP APIs. Therefore you can use the JAXR provider for UDDI to access this registry. You can also use the JAXR provider to access the UDDI registry for UDDI Version 2 in WebSphere Application Server Version 5.x.

To work with the UDDI Version 3 SOAP APIs, use the UDDI Version 3 Client for Java; you cannot use JAXR.

Capability level

The JAXR specification defines two capability profiles, capability level 0 and capability level 1. The JAXR API documentation categorizes each JAXR method as either level 0 or level 1. Generally, a JAXR provider for UDDI has capability level 0 and supports all level 0 methods, whereas a JAXR provider for ebXML has capability level 1 and supports all level 0 and level 1 methods. The JAXR provider for UDDI is a capability level 0 provider, and supports only level 0 methods.

Authentication and security

The javax.xml.registry.authenticationMethod connection property determines which method the JAXR provider uses to authenticate with the UDDI registry.

The JAXR provider uses UDDI Version 2 SOAP Inquiry and Publish APIs. These APIs are protected, as described in the topic about access control for UDDI registry interfaces.

You can use Secure Sockets Layer (SSL) to encrypt HTTP traffic between the JAXR provider for UDDI and the UDDI registry.

Using SSL with the UDDI JAXR provider

You can use Secure Sockets Layer (SSL) to encrypt HTTP traffic between the Java API for XML Registries (JAXR) provider for UDDI and the UDDI registry.

About this task

Note: From WebSphere Application Server Version 8.0, Java API for XML Registries (JAX-R) APIs are deprecated. The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery and Integration (UDDI) 2 technology, which is no longer relevant. If your applications use JAX-R, then you might consider using UDDI 3.

To use SSL, set the JAXR client program as follows.

Procedure

1. For the javax.xml.registry.queryManagerURL and javax.xml.registry.lifeCycleManagerURL connection properties, specify a URL with the protocol https and the appropriate port to use SSL to access the UDDI registry. The default port of the UDDI registry for HTTPS is 9443. Often, only the lifeCycleManager URL, that is, the UDDI Publish API URL, requires SSL.
2. Add a new security provider to the java.security.Security object, according to the Java Secure Sockets Extension (JSSE) implementation that is used. If running under the JVM provided in WebSphere Application Server, the JSSE that is provided by IBM is on the classpath automatically. Use the following code to add the IBM security provider:

```
java.security.Security.addProvider(new com.ibm.jsse.JSSEProvider());
```

3. Set the javax.net.ssl.trustStore system property to the file name of the client trust store file. The client trust store file is a Java key store (.jks) file and must contain the server certificate of the UDDI registry. To manage key store files, you can use the iKeyman tool.
4. Set the javax.net.ssl.trustStorePassword system property. This property is the password used to open the client trust store file.
5. Optional: If you use a JVM version that is earlier than the version that is provided with WebSphere Application Server, you might need to set the java.protocol.handler.pkgs system property to com.ibm.net.ssl.internal.www.protocol.

What to do next

For more information about SSL and the iKeyman tool, see the topic about secure communications using SSL.

Creating a custom internal taxonomy for the JAXR provider

You can create a custom internal taxonomy and make it available to the Java API for XML Registries (JAXR) provider.

About this task

Note: From WebSphere Application Server Version 8.0, Java API for XML Registries (JAX-R) APIs are deprecated. The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery and Integration (UDDI) 2 technology, which is no longer relevant. If your applications use JAX-R, then you might consider using UDDI 3.

The JAXR provider for UDDI supplies a number of internal taxonomies. You can also supply a custom internal taxonomy. To create a new custom internal taxonomy and make it available to the JAXR provider, use the following procedure.

Procedure

1. Create a text file that contains the taxonomy element data. You can use the `iso3166-2003-data.txt` file in `plugins/com.ibm.uddi_1.0.0` as an example. This file is the taxonomy data file for the supplied ISO 3166 taxonomy. The first few lines are:

```
iso3166#--#World#--
iso3166#AD#Andorra#--
iso3166#AE#United Arab Emirates#--
iso3166#AE-AJ#'Ajm?n#AE
iso3166#AE-AZ#Ab? Z?aby[Abu Dhabi]#AE
iso3166#AE-DU#Dubayy [Dubai]#AE
iso3166#AE-FU#Al Fujayrah#AE
iso3166#AE-RK#Ra's al Khaymah#AE
iso3166#AE-SH#Ash Sh?riqah [Sharjah]#AE
iso3166#AE-UQ#Umm al Qaywayn#AE
iso3166#AF#Afghanistan#--
iso3166#AF-BAL#Balkh#AF
iso3166#AF-BAM#B?m??n#AF
```

   Each line represents one element of the taxonomy, or one concept in the taxonomy concept tree. Each line has the following format:

```
<taxonomy ID>#<element value>#<element name>#<parent element value>
```

   The following table describes the tokens in the format.

Table 248. Tokens in the format of the taxonomy element data file. The table lists the different tokens along with a description for each one.

Token	Description
<taxonomy ID>	The taxonomy ID is the same for every element of a taxonomy.
<element value>	The concept value (UDDI keyValue).
<element name>	The concept name (UDDI keyName).
<parent element value>	The value of the parent element of the current element in the taxonomy tree. Except for the root element, for every element in the data file, there must be another line that defines a parent element. The root element is denoted by defining itself as its own parent. There must be only one root element, and no elements without parents.
#	The delimiter character. You can define this character for each taxonomy in the `taxonomyConfig.properties` file; the delimiter does not have to be the number sign (#).

2. Save a ClassificationScheme (UDDI tModel entity) in the UDDI registry to represent the new internal taxonomy. To do this, use the javax.xml.registry.BusinessLifeCycleManager.saveClassificationSchemes() method.
3. Add the new taxonomy to the `taxonomyConfig.properties` file
   a. Copy the supplied `taxonomyConfig.properties` file from the root of the `com.ibm.uddi_1.0.0.jar` file. The content of the supplied `taxonomyConfig.properties` file is:

```
naics-1997 = UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2, naics-1997-data.txt, #
naics-2002 = UUID:1FF729F2-1948-46CF-B660-31EC107F1663, naics-2002-data.txt, #
unspsc  = UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384, unspsc-data.txt,     #
unspsc7_data = UUID:CD153257-086A-4237-B336-6BDCBDCC6634, unspsc7-data.txt,   #
iso3166-2003 = UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88, iso3166-2003-data.txt,#
```

This file has one line for each internal taxonomy that is supplied. Each line has the following format:

```
<taxonomy ID> = <tModelKey>,<data filename>,<data file delimiter>
```

The following table describes the tokens in the format.

Table 249. Tokens in the format of the `taxonomyConfig.properties` file. The table lists the different tokens along with a description for each one.

Token	Description
<taxonomy ID>	The JAXR provider uses this value internally to identify each taxonomy. This value does not have to be the same as the taxonomy ID in the corresponding taxonomy data file.
<tModelKey>	The tModelKey element of the corresponding UDDI tModel entity, which is the id of the corresponding JAXR ClassificationScheme.
<data filename>	The name of the corresponding taxonomy data file.
<data file delimiter>	The delimiter character used in the taxonomy data file. All internal taxonomies that are supplied use the number sign (#), but user-supplied internal taxonomies might use different delimiter characters.

b. Add a new line for the new taxonomy to the copy of the `taxonomyConfig.properties` file. Do not remove any existing taxonomies from the file, because this makes them unavailable to the JAXR provider.

4. Add the copied `taxonomyConfig.properties` file to the Java class path, ahead of the `jaxruddi.jar` file.
5. If there are any JAXR client programs still running that were started before you added the new taxonomy to the `taxonomyConfig.properties` file, create a new connection to pick up the new taxonomy.

JAXR provider for UDDI internal taxonomies

The Java API for XML Registries (JAXR) provider for UDDI supplies a number of internal taxonomies.

Note: From WebSphere Application Server Version 8.0, Java API for XML Registries (JAX-R) APIs are deprecated. The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery and Integration (UDDI) 2 technology, which is no longer relevant. If your applications use JAX-R, then you might consider using UDDI 3.

The following table shows the internal taxonomies that the JAXR provider for UDDI supplies.

Table 250. JAXR provider for UDDI internal taxonomies. The table lists the different internal taxonomies as well as their ClassificationScheme name and ClassificationScheme identification numbers.

Taxonomy	ClassificationScheme name (UDDI tModel name)	ClassificationScheme id (UDDI Version 2 tModelKey)
NAICS 1997	ntis-gov:naics:1997	UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2
NAICS 2002	ntis-gov:naics:2002	UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2
UNSPSC 3.1	unspsc-org:unspsc:3-1	UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384
UNSPSC 7	unspsc-org:unspsc	UUID:CD153257-086A-4237-B336-6BDCBDCC6634
ISO3166 2003	ubr-uddi-org:iso-ch:3166-2003	UUID:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88

The tModel entities that correspond to all these taxonomies are available in the UDDI Version 3 registry. If you use the JAXR provider to access a UDDI Version 2 registry, only the tModel entities that correspond to the NAICS 1997, UNSPSC 3.1, and ISO3166 taxonomies are available.

Each internal taxonomy is loaded into memory once for each JAXR Connection. The classification scheme of the taxonomy is created when the connection is created. At this time, the associated UDDI tModel entity is obtained from the registry and used to populate the ClassificationScheme attributes. The concept object

tree of the taxonomy is not created until the first time the user requests the ClassificationScheme object. All subsequent requests for the same internal taxonomy that use the same connection return the same object tree.

Do not modify any part of the concept object tree programmatically. There is only one classification scheme and concept object tree for each internal taxonomy for each connection, and if you modify the concept tree programmatically, all future requests for this taxonomy that use the same connection return the modified objects, which might not be valid. If you modify the concept tree programmatically, the associated taxonomy data file does not change. To change the values in a user-defined internal taxonomy, change the taxonomy data file, then create a new connection to pick up the changes in a new concept tree.

Similarly, do not modify an internal classification scheme programmatically, except to modify and then save a user-defined internal classification scheme. A new connection is not required to pick up programmatic changes.

JAXR provider logging and messages

The Java API for XML Registries (JAXR) provider for UDDI uses UDDI4J logging, commons logging, and some standard messages.

Note: From WebSphere Application Server Version 8.0, Java API for XML Registries (JAX-R) APIs are deprecated. The Java Platform, Enterprise Edition (Java EE) 6 platform began the deprecation process for JAX-R because it is based on Universal Description, Discovery and Integration (UDDI) 2 technology, which is no longer relevant. If your applications use JAX-R, then you might consider using UDDI 3.

UDDI4J Logging

The JAXR provider for UDDI uses UDDI4J Version 2 to communicate with the UDDI registry. UDDI4J has its own logging. To switch on UDDI4J logging, set the org.uddi4j.logEnabled system property to `true`. The XML request and response bodies of every UDDI request are sent to the standard error log.

Trace Entry, exit, exception, warning, and debug trace is provided by Commons Logging (JCL). Trace is created only if the JAXR client configures it. Entry, exit, and debug trace uses the debug level of logging. Exception and warning trace uses the information level of logging. Additionally, information-level logging is provided before each UDDI4J request is made.

Standard error log messages

The InternalTaxonomyManager, EnumerationManager, and PostalSchemeManager objects send warning messages to the System.err log if there is an error condition that does not justify an exception, but does require an information message. For example, warning messages are sent if a file contains a line that is not valid, or if a tModel entity that corresponds to an internal taxonomy cannot be found in the registry.

Chapter 38. Developing Work area

This page provides a starting point for finding information about work areas, a WebSphere extension for improving developer productivity.

Work areas provide a capability much like that of global variables. They enable efficient sharing of information across a distributed application.

For example, you might want to add profile information as each customer enters your application. By placing this information in a work area, it is available throughout your application, eliminating the need to hand-code a solution or to read and write information to a database.

Developing applications that use work areas

Developing applications that use work areas

Applications interact with the work area service by implementing the UserWorkArea interface. This interface defines all of the methods used to create, manipulate, and terminate work areas.

About this task

```
package com.ibm.websphere.workarea;

public interface UserWorkArea {
   void begin(String name);
   void complete() throws NoWorkArea, NotOriginator;

   String getName();
   String[] retrieveAllKeys();
   void set(String key, java.io.Serializable value)
      throws NoWorkArea, NotOriginator, PropertyReadOnly;
   void set(String key, java.io.Serializable value, PropertyModeType mode)
      throws NoWorkArea, NotOriginator, PropertyReadOnly;
   java.io.Serializable get(String key);
   PropertyModeType getMode(String key);
   void remove(String key)
      throws NoWorkArea, NotOriginator, PropertyFixed;
}
```

Attention: Enterprise JavaBeans (EJB) applications can use the UserWorkArea interface only within the implementation of methods in either the remote or local interface, or both; likewise, servlets can use the interface only within the service method of the HTTPServlet class. Use of work areas within any life cycle method of a servlet or enterprise bean is considered a deviation from the work area programming model and is not supported.

The work area service defines the following exceptions for use with the UserWorkArea interface:

NoWorkArea
: Raised when a request requires an associated work area but none is present.

NotOriginator
: Raised when a request attempts to manipulate the contents of an imported work area.

PropertyReadOnly
: Raised when a request attempts to modify a read-only or fixed read-only property.

PropertyFixed
: Raised by the remove method when the designated property has one of the fixed modes.

Procedure

1. Access a partition by either:
   - Accessing the UserWorkArea partition, to access the UserWorkArea partition.

- Accessing a user-defined work area partition, to access a user-defined work area.

The following steps use the UserWorkArea partition as an example; however, you can use a user-defined partition in the same way.

2. Begin a new work area.

   Use the begin method to create a new work area and associate it with the calling thread. A work area is scoped to the thread that began the work area and is not accessible by multiple threads. The begin method takes a string as an argument; the string is used to name the work area. The argument must not be null, which causes the java.lang.NullPointer exception to be raised. In the following code example, the application begins a new work area with the name SimpleSampleServlet:

```
public class SimpleSampleServlet {
...
   try {
      ...
      userWorkArea = (UserWorkArea)jndi.lookup(
         "java:comp/websphere/UserWorkArea");
   }
   ...

   userWorkArea.begin("SimpleSampleServlet");
   ...
}
```

   The begin method is also used to create nested work areas; if a work area is associated with a thread when the begin method is called, the method creates a new work area nested within the existing work area.

   The work area service makes no use of the names associated with work areas; You can name work areas in any way that you choose. Names are not required to be unique, but the usefulness of the names for debugging is enhanced if the names are distinct and meaningful within the application. Applications can use the getName method to return the name associated with a work area by the begin method.

3. Set properties in the work area.

   An application with a current work area can insert properties into the work area and retrieve the properties from the work area. The UserWorkArea interface provides two set methods for setting properties and a get method for retrieving properties. The two-argument set method inserts the property with the property mode of normal. The three-argument set method takes a property mode as the third argument. Both set methods take the key and the value as arguments. The key is a String; the value is an object of the type java.io.Serializable. None of the arguments can be null, which causes the java.lang.NullPointer exception to be raised.

   The SimpleSample application below uses objects of two classes, the SimpleSampleCompany class and the SimpleSampleProperty class, as values for properties. The SimpleSampleCompany class is used for the site identifier, and the SimpleSamplePriority class is used for the priority. These classes are shown in following code example:

```
public class SimpleSampleServlet {
   ...
   userWorkArea.begin("SimpleSampleServlet");

   try {
      // Set the site-identifier (default is Main).
      userWorkArea.set("company",
         SimpleSampleCompany.Main, PropertyModeType.read_only);

      // Set the priority.
      userWorkArea.set("priority", SimpleSamplePriority.Silver);
   }

   catch (PropertyReadOnly e) {
     // The company was previously set with the read-only or
     // fixed read-only mode.
     ...
   }

    catch (NotOriginator e) {
```

```
      // The work area originated in another process,
      // so it can't be modified here.
      ...
   }

   catch (NoWorkArea e) {
      // There is no work area begun on this thread.
      ...
   }

   // Do application work.
   ...
}
```

The get method takes the key as an argument and returns a Java Serializable object as the value associated with the key. For example, to retrieve the value of the company key from the work area, the code example above uses the get method on the work area to retrieve the value.

Setting property modes. The two-argument set method on the UserWorkArea interface takes a key and a value as arguments and inserts the property with the default property mode of normal. To set a property with a different mode, applications must use the three-argument set method, which takes a property mode as the third argument. The values used to request the property modes are as follows:

- **Normal**: PropertyModeType.normal
- **Fixed normal**: PropertyModeType.fixed_normal
- **Read-only**: PropertyModeType.read_only
- **Fixed read-only**: PropertyModeType.fixed_readonly

4. Manage local work with a work area.
5. Complete the work area.

   After an application has finished using a work area, it must complete the work area by calling the complete method on the UserWorkArea interface. This terminates the association with the calling thread and destroys the work area. If the complete method is called on a nested work area, the nested work area is terminated and the parent work area becomes the current work area. If there is no work area associated with the calling thread, a NoWorkArea exception is created. Every work area must be completed, and work areas can be completed only by the originating process. For example, if a server attempts to call the complete method on a work area that originated in a client, a NotOriginator exception is created. Work areas created in a server process are never propagated back to an invoking client process.

   Attention: The work area service claims full local-remote transparency. Even if two beans happen to be deployed in the same server, and therefore the same JVM and process, a work area begun on an invocation from another is completed and the bean in which the request originated is always in the same state after any remote call.

   The following code example shows the completion of the work area created in the client application.

```
public class SimpleSampleServlet {
   ...
   userWorkArea.begin("SimpleSampleServlet");
   userWorkArea.set("company",
       SimpleSampleCompany.Main, PropertyModeType.read_only);
   userWorkArea.set("priority", SimpleSamplePriority.Silver);
   ...

   // Do application work.
   ...

   // Terminate the work area.
   try {
      userWorkArea.complete();
   }

   catch (NoWorkArea e) {
      // There is no work area associated with this thread.
      ...
   }

   catch (NotOriginator e) {
      // The work area was imported into this process.
```

```
        ...
      }
     ...
  }
```

The following code example shows the sample application completing the nested work area that it created earlier in the remote invocation.

```
public class SimpleSampleBeanImpl implements SessionBean {

    public String [] test() {
      ...

      // Begin a nested work area.
      userWorkArea.begin("SimpleSampleBean");
      try {
        userWorkArea.set("company",
                         SimpleSampleCompany.London_Development);
      }
      catch (NotOriginator e) {
      }

      SimpleSampleCompany company =
         (SimpleSampleCompany) userWorkArea.get("company");
      SimpleSamplePriority priority =
         (SimpleSamplePriority) userWorkArea.get("priority");

      // Complete all nested work areas before returning.
      try {
        userWorkArea.complete();
      }
      catch (NoWorkArea e) {
      }
      catch (NotOriginator e) {
      }
   }
}
```

Example

- Work area object types
- Using the work area partition manager

Work area object types. In the following example, the client creates a work area and inserts two properties into the work area: a site identifier and a priority. The site-identifier is set as a read-only property; the client does not allow recipients of the work area to override the site identifier. This property consists of the key company and a static instance of a SimpleSampleCompany object. The priority property consists of the key priority and a static instance of a SimpleSamplePriority object. The object types are defined as shown in the following code example:

```
public static final class SimpleSampleCompany {
   public static final SimpleSampleCompany Main;
   public static final SimpleSampleCompany NewYork_Sales;
   public static final SimpleSampleCompany NewYork_Development;
   public static final SimpleSampleCompany London_Sales;
   public static final SimpleSampleCompany London_Development;
}

public static final class SimpleSamplePriority {
   public static final SimpleSamplePriority Platinum;
   public static final SimpleSamplePriority Gold;
   public static final SimpleSamplePriority Silver;
   public static final SimpleSamplePriority Bronze;
   public static final SimpleSamplePriority Tin;
}
```

The client then makes an invocation on a remote object. The work area is automatically propagated; none of the methods on the remote object take a work area argument. On the remote side, the request is first handled by the SimpleSampleBean; the bean first reads the site identifier and priority properties from the

work area. The bean then intentionally attempts, and fails, both to write directly into the imported work area and to override the read-only site-identifier property.

The SimpleSampleBean successfully begins a nested work area, in which it overrides the client's priority, then calls another bean, the SimpleSampleBackendBean. The SimpleSampleBackendBean reads the properties from the work area, which contains the site identifier set in the client and priority set in the SimpleSampleBean. Finally, the SimpleSampleBean completes its nested work area, writes out a message based on the site-identifier property, and returns.

Using the work area partition manager. The following code example illustrates the use of the work area partition manager interface. The sample illustrates how to create and retrieve a work area partition programmatically`. Please note that programmatically creating a work area partition is only available on the Java Platform, Enterprise Edition (Java EE) client. To create a work area partition on the server one must use the administrative console. Refer to the Work area partition service article for configuration parameters available to configure a partition.

```
import com.ibm.websphere.workarea.WorkAreaPartitionManager;
import com.ibm.websphere.workarea.UserWorkArea;
import com.ibm.websphere.workarea.PartitionAlreadyExistsException;
import com.ibm.websphere.workarea.NoSuchPartitionException;
import java.lang.IllegalAccessError;
import java.util.Properties;
import javax.naming.InitialContext;

//This sample demonstrates how to retrieve an instance of the
//WorkAreaPartitionManager implementation and how to use that
//instance to create a WorkArea partition and retrieve a partition.
//NOTE: Creating a partition in the way listed below is only available
//on a J2EE client.  To create a partition on the server use the
//WebSphere administrative console.  Retrieving a WorkArea
//partition is performed in the same way on both client and server.

public class Example {

     //The name of the partition to create/retrieve
     String partitionName = "myPartitionName";
     //The name in java naming the WorkAreaPartitionManager instance is bound to
     String jndiName = "java:comp/websphere/WorkAreaPartitionManager";

     //On a J2EE client a user would create a partition as follows:
     public UserWorkArea myCreate(){
         //Variable to hold our WorkAreaPartitionManager reference
         WorkAreaPartitionManager partitionManager = null;
         //Get an instance of the WorkAreaPartitionManager implementation
         try {
             InitialContext initialContext = new InitialContext();
             partitionManager = (WorkAreaPartitionManager) initialContext.lookup(jndiName);
         } catch (Exception e) {  }

         //Set the properties to configure our WorkArea partition
         Properties props = new Properties();
         props.put("maxSendSize","12345");
         props.put("maxReceiveSize","54321");
         props.put("Bidirectional","true");
      props.put("DeferredAttributeSerialization","true");

         //Variable used to hold the newly created WorkArea Partition
         UserWorkArea myPartition = null;

         try{
             //This is the way to create a partition on the J2EE client.  Use the
             //WebSphere Administrative Console to create a WorkArea Partition
             //on the server.
             myPartition = partitionManager.createWorkAreaPartition(partitionName,props);
         }
         catch (PartitionAlreadyExistsException e){  }
         catch (IllegalAccessException e){  }
```

```
        return myPartition;
    }

    //. . . .

    //In order to retrieve a WorkArea partition at some time later or
    //from some other class, do the following (from client or server):
    public UserWorkArea myGet(){
        //Variable to hold our WorkAreaPartitionManager reference
        WorkAreaPartitionManager partitionManager = null;
        //Get an instance of the WorkAreaPartitionManager implementation
        try {
            InitialContext initialContext = new InitialContext();
            partitionManager = (WorkAreaPartitionManager) initialContext.lookup(jndiName);
        } catch (Exception e) {  }

        //Variable used to hold the retrieved WorkArea partition
        UserWorkArea myPartition = null;
        try{
            myPartition = partitionManager.getWorkAreaPartition(partitionName);
        }catch(NoSuchPartitionException e){  }

        return myPartition;
    }
}
```

What to do next

For additional information about work area, see the com.ibm.websphere.workarea package in the API. The generated API documentation is available in the information center table of contents from the path **Reference** > **APIs - Application programming interfaces**.

Configuring work area partitions

Configuring work area partitions

About this task

The work area partition service extends the work area service by allowing the creation of multiple work areas with more configuration options than what is available to the UserWorkArea partition. Follow these steps to create and configure a work area partition:

Procedure

1. Create a user defined partition on the server.
   a. Start the administrative console.
   b. Click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **Business process services** > **Work area partition service**.
   c. Click **New**.
   d. On the settings page for work area partitions, specify values such as the partition name, maxSendSize and maxReceiveSize, then click OK.
   e. Save the new configuration.
   f. Restart the server to apply the new configuration.
2. Create a user defined partition on the client. Use the createWorkAreaPartition method described in the The Work area partition manager interface article to programmatically create a partition. Refer to the Example: Using the work area partition manager article for an example of using this method.

Results

You have created a work area partition.

What to do next

Retrieve the partition through the work area partition manager interface and use it as defined by the work area service and the UserWorkArea interface. Refer to the Example: Using the work area partition manager article for an example.

Work area partition service

The work area partition service is an extension of the work area service that enables the creation of multiple custom work areas. The work area partition service is an optional service to users. Any user that currently uses the work area service and the UserWorkArea partition can continue using it in the same manner. The UserWorkArea partition is created automatically (if it has not been disabled) by the work area partition service. By allowing a user the option to create their own work area partition through the work area partition service, they can have more control over configuration and access to their partition.

The work area partition service is essentially a factory for creating instances of the UserWorkArea interface. Applications interact with work areas by using the UserWorkArea interface and its implementation. This interface defines all of the methods used to create, manipulate, and complete work areas. The work area partition service allows users to create their own named instance of the UserWorkArea interface. Each named instance is called a user-defined work area partition, or partition for short. Each instance of the UserWorkArea interface (partition) is separate from other user-defined partitions. Furthermore, you can configure a partition with various options to provide qualities of service that are unique to a use case for an individual user. Any configuration option made within the work area partition service panel does not affect the work area service.

Unlike the UserWorkArea partition, which is publicly known, work areas created by the work area partition service are accessible to, and known only by, the creator. However, the work area partition service does not strictly enforce that a partition is accessed and/or operated on exclusively by the partition creator. There are no limitations should the creator want to publish their work area partition and make it publicly available by binding their partition reference in Java naming or by other means. However, the work area partition service does try to hide a partition as much as possible should a user not want others to know about a certain partition. The work area partition service does not enable a person to determine or query the names of all the partitions that have been created; however, it does not restrict the partitions from being accessed by users other than the creator of that partition. The context of a partition, such as the UserWorkArea partition or a user-defined partition, is scoped to a single thread and is not accessible by multiple threads.

The work area partition reference that is returned to a user implements javax.naming.Referenceable and com.ibm.websphere.UserWorkArea, therefore a user can bind their partition into a name to make their partition publicly available. An alternative to using Java naming to bind and access the partition is to use the work area partition manager interface. Anyone can access the work area partition manager interface; therefore, if a user wants to make their partition publicly available, they simply need to publish their partition name. Other users can then call the getWorkAreaPartition method on the work area partition manager interface with the published name.

The WorkAreaPartitionManager.createWorkAreaPartition method can only be used from a Java Platform, Enterprise Edition (Java EE) client. To create a work area partition on the server side, one must use the administrative console. On the server side a work area partition must be created during server startup because each partition needs to be register with the appropriate Web and Enterprise JavaBeans (EJB) collaborators before the server has started. Custom work area partitions are created by the work area partition service and defined by the UserWorkArea interface.

The work area partition service also allows a user to configure partitions with additional properties that are not available on the UserWorkArea partition, such as bidirectional propagation of work area partition context and deferred attribute serialization. These properties are available as configuration properties when creating a partition. For a complete list of the configuration properties that are available when creating a

partition, see the "Configurable Work Area Partition Properties" section in the Work area partition manager interface article. The properties are defined as follows:

Bidirectional propagation of work area context

If a remote invocation is issued from a thread associated with a work area, a copy of the work area is automatically propagated to the target object, which can use or ignore the information in the work area as necessary. If the calling application has a nested work area associated with it, a copy of the nested work area and all its ancestors are propagated to the target application. The target application can locally modify the information, as allowed by the property modes, by creating additional nested work areas; this information is propagated to any remote objects that it invokes.

Whether context changes propagate back to a calling application from a remote application depends on the configuration of the work area partition. If a user creates a partition to be bidirectional by selecting the Bidirectional property during partition creation, changes made by a remote application propagate back to the calling application, meaning that changes made to the work area context by a downstream process will propagate back upstream. The UserWorkArea partition is not configured (and can never be configured) to be bidirectional, therefore context changes only flow to downstream processes and do not propagate back upstream.

Example: Bidirectional propagation of work area context

Whether context changes propagate back to a calling application from a remote application depends on the configuration of the work area partition. If a user creates a bidirectional partition, changes made by a remote application propagate back to the calling application. Changes made to the work area context by a downstream process propagate back upstream. The figure Distribution of work area context when configured for bidirectional propagation illustrates this relationship during a remote call to the server. For this illustration, the client and server must have created a partition with the same name.

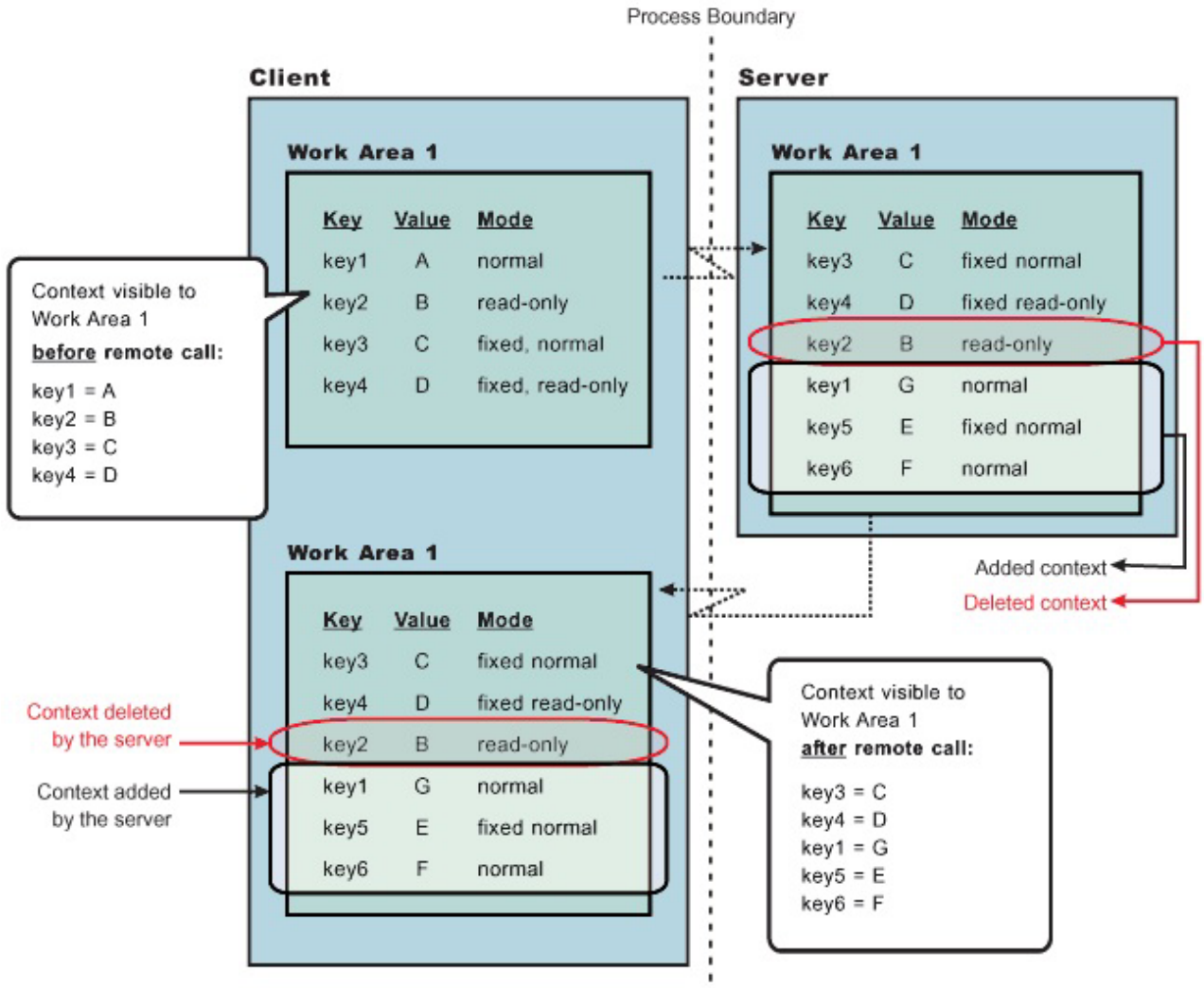


Figure 7. Distribution of work area context when configured for bidirectional propagation. This figure illustrates the distribution of work area context when the service is configured for bidirectional propagation.

When the client makes a remote call to the server, the server receives the context set by the client process. The server then can make changes to this context or add to it. In this illustration, the server overwrites the value at **key1**, removes the property at **key2**, and adds two new properties at **key5** and **key6**. When the server application returns to the client, the work area context is propagated back to the client and unmarshalled. The current work area is then updated with the new context. Note that if the partition is not configured as bidirectional and the server tries to change or remove context in Work Area 1, it receives a com.ibm.websphere.workarea.NotOriginator exception because the client was the originator of the work area. The server can retrieve the context in Work Area 1. This is the main distinction between bidirectional propagation of context and non-bidirectional propagation.

Example: Bidirectional propagation of nested work area context

If a remote application needs to add context to a work area that is only used by itself or any other remote objects, the remote application must begin another work area. By beginning a new work area, the additional context is scoped to that application and does not flow back to the calling application. The major benefit of nesting work areas is that nesting work areas allows an application to scope work area context to a given application. Taking the illustration one step further, if the server has begun a work area before overwriting the value at **key1**, removing the property at **key2**, or adding new properties at **key5** and **key6**;

those changes would not have propagated back to the client. This is shown in the figure Distribution of nested work area context when configured for bidirectional propagation. You can also see from this figure that the client does not receive the context from the nested work area started by the server.

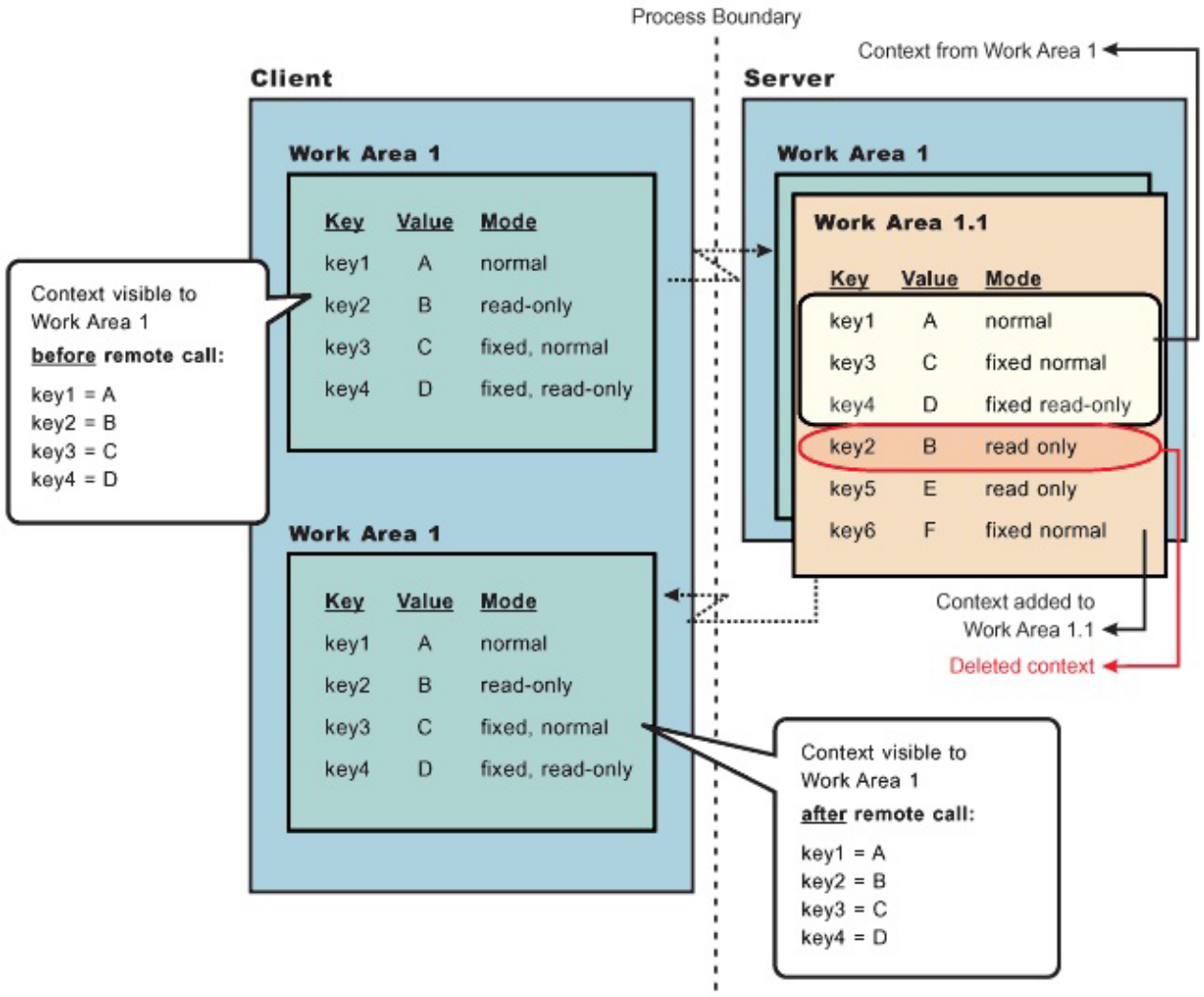


Figure 8. Distribution of nested work area context when configured for bidirectional propagation. This figure illustrates the distribution of nested work area context when the service is configured for bidirectional propagation.

Deferred attribute serialization of work area context

By default, on each set operation the attribute set into a work area is automatically serialized by the work area service. On each subsequent get operation on that same attribute it is deserialized and returned to the requester. This gives the work area service complete control of the attribute such that any changes to a mutable object are not reflected in the work area's copy of the attribute unless a user specifically resets the attribute into the work area. However, this can potentially lead to excessive serialization and deserialization.

Excessive serialization and deserialization can result in observable performance degradation under heavy load. The deferred attribute serialization configuration property is a caching feature that reduces serialization and deserialization operations. When deferred attribute serialization is enabled in a client or server process by selecting the Deferred Attribute Serialization field during the creation of the work area, attributes set into the work area service are not automatically serialized during the set operation. Rather, a

reference to the attribute is stored in the work area. If the attribute is mutable, then changes to the object are reflected in the work area's reference to that attribute. When a get operation is performed on that attribute, the reference to that object is returned and no deserialization is performed.

Attributes are not serialized until the thread with which the attribute is associated makes a remote IIOP invocation. At that point, the attribute is serialized and the serialized form of the attribute is cached. If the attribute is not reset into the work area, changes to the original attribute are still reflected within the attribute contained within the work area because the work area still holds a cached reference to the original object. However, if the work area has not been told that the attribute has changed by resetting the attribute into the work area, subsequent remote requests continue to use the cached serialized version of the attribute and direct changes to the mutable attribute are not propagated. This is an important distinction between enabling and not enabling the deferred attribute serialization configuration property, and a user must pay close attention to this difference and how mutable objects are handled when enabling deferred attribute serialization. The work area service releases cached references and cached serialized versions of attributes when any of the following conditions occur:

- An attribute is reset or removed.
- The work area is explicitly completed by the application.
- Server component ends execution of the request during which the work area was begun.
- Client process which began the work area terminates.

Partition context propagation across process boundaries

Work area context automatically propagates from client to server when a client makes a remote call to a server. For example, if a client is configured with three different work area partitions when it makes a remote call to a server, server1, the context associated with each partition on the client thread propagates to server1. If the same three partitions have been created on server1, the context is unmarshalled to the appropriate partition. However, if none or only a few of the three partitions have been created on server1, only the context associated with a partition that is resident on both the client and server is unmarshalled. The context associated with a partition that is not resident on server1 is still resident on server1 but is not accessible. The context associated with partitions that are not resident on server1 must remain resident on server1 in case another remote call is made to a different server. Going one step further, if server1 makes a call to another server, server2, that has the same partitions as the client, server2 receives the context for the partitions that were not resident on server1. Any partitions that reside on server1 that did not reside on the client now have their context propagated to server2.

For more information about work area, see the com.ibm.websphere.workarea package in the Application Programming Interface (API). The generated API documentation is available in the information center table of contents from the path **Reference** > **APIs - Application Programming Interfaces**.

The Work area partition manager interface

Applications interact with the work area partition service by using the work area partition manager interface. A user can retrieve an instance of the work area partition manager interface out of naming and use the methods that are defined in the following section.

An implementation of the work area partition manager interface is bound in Java naming at `java:comp/websphere/WorkAreaPartitionManager`. This interface is responsible for creating, retrieving, and manipulating work area partitions:

```
package com.ibm.websphere.workarea;

import com.ibm.websphere.workarea.UserWorkArea;
import com.ibm.websphere.workarea.PartitionAlreadyExistsException;
import com.ibm.websphere.workarea.NoSuchPartitionException;
import java.util.Properties;

public interface WorkAreaPartitionManager {
```

```
  //Returns an instance of a work area partition for the given name, or throws an exception if the
  //partition name doesn't exists.
  public UserWorkArea getWorkAreaPartition(String partitionName) throws NoSuchPartitionException;

  //Returns a new instance of a work area partition (an implementation of the UserWorkArea interface)
  //or throws an exception if the partition name already exists.  The createWorkAreaPartition should
  //only be used within a Java EE platform client and NOT on the
  //server.  To create a work area partition on the server, use the WebSphere administrative
  //console.
  public UserWorkArea createWorkAreaPartition(String partitionName, Properties props) throws
     PartitionAlreadyExistsException, java.lang.IllegalAccessException;
  }
}
```

EJB applications can use the work area partition manager interface only within the implementation of methods in either the remote or local interface, or both; likewise, servlets can use the interface only within the service method of the HTTPServlet class. Use of work areas within any life cycle method of a servlet or enterprise bean is considered a deviation from the work area programming model and is not supported.

Programmatically creating a work area partition through the createWorkAreaPartition method is only available on the Java EE client. To create a work area partition on the server, use the WebSphere administrative console as described in the Configuring work area partitions article. All partitions in a server process must be created before server startup is complete so that the work area service can register with the appropriate container collaborators. Therefore, calling the createWorkAreaPartition method in a server process after the server starts results in a java.lang.IllegalAccessException exception. The createWorkAreaPartition method can be called in a Java EE application client at any time.

Configurable Work Area Partition Properties

This section applies to the use of the createWorkAreaPartition method on the WorkAreaPartitionManager interface. As is described above, this method should only be used on a Java EE client. To create a partition on the server, please see Configuring work area partitions.

The "createWorkAreaPartition" method on the WorkAreaPartitionManager interface takes a java.util.Properties objects. This Properties object, and the properties it contains, is used to define the work area partition. Below is an example of creating a Properties object and setting a property:

Attention: A more detailed example of the usage of the WorkAreaPartitionManager can be found in Example: Using the work area partition manager.

```
java.util.Properties props = new java.util.Properties():
props.put("maxSendSize","12345");
```

Acceptable key/values pairs (properties) for defining a partition are as follows:

- *maxSendSize* - Indicates the maximum size (bytes) of a work area that can be sent on a remote call. Acceptable values are:
  - "-1" = Uses the default size of 32767.
  - "0" = Unlimited size, this value will not be policed which might help performance a bit depending on the number of work area an application has.
  - "1" = Integer.MAX_VALUE
- *maxReceiveSize* - Indicates the maximum size (bytes) of a work area that can be received. Acceptable values are:
  - "-1" = Uses the default size of 32767.
  - "0" = Unlimited size, this value will not be policed which might help performance a bit depending on the number of work area an application has.
  - "1" = Integer.MAX_VALUE

- *Bidirectional* - Indicates if work area context that is changed by a downstream process should be propagated back upstream to the originator of that context. For a more complete description of this property, refer to the "Bidirectional propagation of work area context" in the Work area partition service article. Acceptable values are:
  - "true" = Context changes will be returned from a remote call.
  - "false" = Context changes will not be returned from a remote call.

  Attention: The default setting is "false."
- *DeferredAttributeSerialization* - Indicates if the serialization of attribute should be optimized to occur exactly once per process. For a more complete description of this property, refer to the "Deferred attribute serialization of work area context" section in the Work area partition service article. Acceptable values are:
  - "true"
    - When an attribute is set into the work area, it will not be serialized until a remote request is made.
    - If the value is unchanged by response, the serialized form will be used for subsequent requests; the live object will be retrieved via getters.
    - When requests are made during a remote request, a value is deserialized on demand exactly once. The serialized form is used for subsequent requests from this remote process on this distributed thread; subsequent requests in process for the same attribute returns the already deserialized value. There are risks with concurrency with DeferredAttributeSerialization. After serialization in a client process, updates to the attribute are no longer reflected in the work area's copy until the value is explicitly reset through the UserWorkArea interface. Changes made to a retrieved reference in a downstream process are not propagated to subsequent downstream requests (or returned on the reply as a changed value) unless explicitly reset through the UserWorkArea interface.
  - "false"
    - When an attribute is set into the work area, it is immediately serialized and the bytes are stored.
    - When an attribute is retrieved from the work area, it is always deserialized from stored bytes.

  Attention: The default value is "false."
- *EnableWebServicePropagation* - Indicates if work area context must propagate on a WebService call. Acceptable values are:
  - "true" = Context propagates on a WebService call.
  - "false" = Context does not propagate on a WebService call.

  Attention: The default value is "false."

Exceptions

The work area partition service defines the following exceptions for use with the work area partition manager interface:

PartitionAlreadyExistsException
: This exception is raised by the createWorkAreaPartition method on the WorkAreaPartitionManager implementation if a user tries to create a work area partition with a partition name that already exists. Partition names must be unique.

NoSuchPartitionException
: This exception is raised by the getWorkAreaPartition method on the WorkAreaPartitionManager implementation if a user requests a work area partition with a partition name that does not exist.

java.lang.IllegalAccessException
: This exception is raised by the createWorkAreaPartition method on the WorkAreaPartitionManager implementation if a user tries to create a work area partition during run time on a server process. This method can only be used on a Java EE client process. In the server process, a partition must be created using the administrative console.

For additional information about work area, see the com.ibm.websphere.workarea package in the application programming interface (API). The generated API documentation is available in the information center table of contents from the path **Reference** > **APIs - Application Programming Interfaces**.

Example: Using the work area partition manager

The example below demonstrates the use of the work area partition manager interface. The sample illustrates how to create and retrieve a work area partition programmatically`. Please note that programmatically creating a work area partition is only available on the Java Platform, Enterprise Edition (Java EE) client. To create a work area partition on the server one must use the administrative console. Refer to the Work area partition service article for configuration parameters available to configure a partition.

```
import com.ibm.websphere.workarea.WorkAreaPartitionManager;
import com.ibm.websphere.workarea.UserWorkArea;
import com.ibm.websphere.workarea.PartitionAlreadyExistsException;
import com.ibm.websphere.workarea.NoSuchPartitionException;
import java.lang.IllegalAccessError;
import java.util.Properties;
import javax.naming.InitialContext;

//This sample demonstrates how to retrieve an instance of the
//WorkAreaPartitionManager implementation and how to use that
//instance to create a WorkArea partition and retrieve a partition.
//NOTE: Creating a partition in the way listed below is only available
//on a J2EE client.  To create a partition on the server use the
//WebSphere administrative console.  Retrieving a WorkArea
//partition is performed in the same way on both client and server.

public class Example {

     //The name of the partition to create/retrieve
     String partitionName = "myPartitionName";
     //The name in java naming the WorkAreaPartitionManager instance is bound to
     String jndiName = "java:comp/websphere/WorkAreaPartitionManager";

     //On a J2EE client a user would create a partition as follows:
     public UserWorkArea myCreate(){
         //Variable to hold our WorkAreaPartitionManager reference
         WorkAreaPartitionManager partitionManager = null;
         //Get an instance of the WorkAreaPartitionManager implementation
         try {
             InitialContext initialContext = new InitialContext();
             partitionManager = (WorkAreaPartitionManager) initialContext.lookup(jndiName);
         } catch (Exception e) {  }

         //Set the properties to configure our WorkArea partition
         Properties props = new Properties();
         props.put("maxSendSize","12345");
         props.put("maxReceiveSize","54321");
         props.put("Bidirectional","true");
      props.put("DeferredAttributeSerialization","true");

         //Variable used to hold the newly created WorkArea Partition
         UserWorkArea myPartition = null;

         try{
             //This is the way to create a partition on the J2EE client.  Use the
             //WebSphere Administrative Console to create a WorkArea Partition
             //on the server.
             myPartition = partitionManager.createWorkAreaPartition(partitionName,props);
         }
         catch (PartitionAlreadyExistsException e){  }
         catch (IllegalAccessException e){  }
```

```
        return myPartition;
    }

    //. . . .

    //In order to retrieve a WorkArea partition at some time later or
    //from some other class, do the following (from client or server):
    public UserWorkArea myGet(){
        //Variable to hold our WorkAreaPartitionManager reference
        WorkAreaPartitionManager partitionManager = null;
        //Get an instance of the WorkAreaPartitionManager implementation
        try {
            InitialContext initialContext = new InitialContext();
            partitionManager = (WorkAreaPartitionManager) initialContext.lookup(jndiName);
        } catch (Exception e) {  }

        //Variable used to hold the retrieved WorkArea partition
        UserWorkArea myPartition = null;
        try{
            myPartition = partitionManager.getWorkAreaPartition(partitionName);
        }catch(NoSuchPartitionException e){  }

        return myPartition;
    }
}
```

Work area partition collection

Use this page to manage the work area service.

The work area partition service supports the definition of custom work area partitions.

To view this administrative console page, click **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **Business Process Services** > **Work area partition service**.

Name

Specifies the name of the work area partition that is used to retrieve the partition. This name must be unique.

Description

Specifies the description of the work area partition.

Enable service at server startup

Specifies whether the server attempts to start the specified service when the server starts.

Bidirectional

Permits applications to modify the context of a work area that is imported by a Java EE request; modified properties are propagated back to the requestor environment. This option is disabled by default.

Maximum send size

Specifies the maximum size of data that can be sent within a single work area. (0 = no limit; -1 = default)

A value of 0 means there is no limit to the data sent. The default value of -1 represents 32768 bytes of data sent.

Maximum receive size

Specifies the maximum size of data that can be received within a single work area. (0 = no limit; -1 = default)

A value of 0 means there is no limit to the data received. The default value of -1 represents 32768 bytes of data received.

Deferred attribute serialization

Specifies whether attribute serialization is deferred until the work area is propagated on a remote invocation.

Enable Web service propagation

Specifies whether the work area partition is propagated on Web service requests. This option is disabled by default.

Work area partition settings

Use this page to modify the work area partition settings.

The work area partition service supports the definition of custom work area partitions.

To view this administrative console page, click **Servers > Server types > WebSphere application servers >** *server_name*. Under **Container Settings**, expand **Business Process Services** then click **Work area partition service**. Click on a *work_area_partition_name*.

Name

Specifies the name of the work area partition that is used to retrieve the partition. This name must be unique.

Description

Specifies the description of the work area partition.

Enable service at server startup

Specifies whether the server attempts to start the specified service when the server starts.

Bidirectional

Permits applications to modify the context of a work area that is imported by a Java EE request; modified properties are propagated back to the requestor environment. This option is disabled by default.

Maximum send size

Specifies the maximum size of data that can be sent within a single work area. (0 = no limit; -1 = default)

Information	Value
Data type	Integer
Units	Bytes
Default	32768
Range	-1, 0 (no limit) and 1 to 2147483647

Maximum receive size

Specifies the maximum size of data that can be received within a single work area. (0 = no limit; -1 = default)

Information	Value
Data type	Integer
Units	Bytes

Information	Value
Default	32768
Range	-1, 0 (no limit) and 1 to 2147483647

Deferred attribute serialization
Specifies whether attribute serialization is deferred until the work area is propagated on a remote invocation. This option is disabled by default.

Enable Web service propagation
Specifies whether the work area partition is propagated on Web service requests. This option is disabled by default.

Accessing a user defined work area partition

About this task

The work area partition service provides a Java Naming and Directory Interface (JNDI) binding to an implementation of the work area partition manager interface under the name java:comp/websphere/WorkAreaPartitionManager. Applications that need to access their partition can perform a lookup on that JNDI name and then use the getWorkAreaPartition method on the work area partition manager, as shown in the following code example:

Example

```
import com.ibm.websphere.workarea.*;
import javax.naming.*;

public class SimpleSampleServlet {
  ...

        //Variable to hold our WorkAreaPartitionManager implementation
        WorkAreaPartitionManager partitionManager = null;
        try {
            InitialContext initialContext = new InitialContext();
            partitionManager = (WorkAreaPartitionManager)
           initialContext.lookup("java:comp/websphere/WorkAreaPartitionManager");
        } catch (Exception e) {...}

        //Variable used to hold the retrieved WorkArea Partition
        UserWorkArea myPartition = null;
        try{
            myPartition = partitionManager.getWorkAreaPartition(partitionName);
        }catch(NoSuchPartitionException e){...}
}
```

What to do next

The next step is to use the begin method to create a new work area and associate it with the calling thread, as described in the Beginning a new work area topic.

Propagating work area context over Web services

WebSphere Application Server Version 6.1 introduces the option to propagate work area context on a Web service call. Prior to WebSphere Application Server Version 6.1, work area context was only propagated over RMI/IIOP calls. The work area application programming interfaces (APIs) have not changed to implement this propagation. You can use the work area APIs as they have in the past and as outlined in the work area documentation. However, by default, work area context is not propagated on a Web service call, you must enable this option.

Procedure

1. Enable a server to propagate work area context on a Web service call.
   a. Start the administrative console.
   b. Select **Servers** > **Server Types** > **WebSphere application servers** > ***server_name*** > **Business Process Services** .
      - To enable the work area service, (the UserWorkArea partition) to propagate its context on a Web service call:
        – Select **Work area service**.
      - To enable an individual partition to propagate its context on a Web service call:
        – Select **Work area partition service**.
        – Select a partition.
   c. Check the EnableWebServicePropagation field to enable Web service propagation.
   d. Save the new configuration and restart the server to apply the new configuration.
2. Enable a client to propagate work area context on a Web service call:

   Note: The steps below are for the work area service (the UserWorkArea partition). For user defined partitions the EnableWebServicePropagation property must be set when creating a partition on the client, refer to the The Work area partition manager interface article.

   a. Set the property com.ibm.websphere.workarea.EnableWebServicePropagation to true when invoking the launchClient script found in the $WAS_HOME/bin directory. For example, to set this property to true, add the following system properties to the launchClient invocation as needed: -CCDcom.ibm.websphere.workarea.EnableWebServicePropagation=true
   b. Set the property com.ibm.websphere.workarea.EnableWebServicePropagation in a property file that is used by the launchClient script. Refer to the Running a Java EE client application with launchClient for additional information.

Chapter 39. XML applications

This page provides a starting point for finding information about XML applications.

Overview of XML support

You can use the XML support provided with this product to work with web applications that process data using standard XML technologies like Extensible Stylesheet Language Transformations (XSLT), XML Path Language (XPath), and XML Query Language (XQuery).

XML-structured data has become the predominant format for data interchange. XML data is navigated, queried, or transformed in almost every existing WebSphere application.

Since first being standardized, XML usage in application-development environments has grown significantly to include many scenarios. WebSphere Application Server is a leading platform for the latest application development standards, including XML.

Note: IBM WebSphere Application Server Version 8.5 delivers critical technology that provides application developers with support for the following key World Wide Web Consortium (W3C) XML standards:

- Extensible Stylesheet Language Transformations (XSLT) 2.0
- XML Path Language (XPath) 2.0
- XML Query Language (XQuery) 1.0

These new and updated W3C XML standards offer application developers numerous advanced capabilities for building XML applications. Specific benefits delivered in the XPath 2.0, XSLT 2.0, and XQuery 1.0 standards include the following:

- Simpler XML application development and improved developer productivity
- Improved ability to query large amounts of data stored in XML outside of a database with XQuery 1.0
- Improved XML-application performance through new features introduced in the W3C specifications to address previous shortcomings
- Improved XML-application reliability with new support for XML schema-aware processing and validation

Note: If you want to use XPath 1.0 or XSLT 1.0 (not in backwards-compatibility mode), continue to use Java API for XML Processing (JAXP) in Java 2 Platform, Standard Edition (J2SE) 6.0 and 7.0.

For more information about these W3C XML standards, go to W3C XQuery 1.0 and XSLT 2.0 Become Standards: Tools to Query, Transform, and Access XML and Relational Data.

The product provides the IBM XML Application Programming Interface in support of these standards. This application programming interface invokes a runtime engine that is capable of executing XPath 2.0, XSLT 2.0, and XQuery 1.0 as well as manipulating the returned XML data.

The product also includes the IBM Thin Client for XML with WebSphere Application Server. The thin client allows access to the same XML API and runtime functionality (XPath 2.0, XSLT 2.0, and XQuery 1.0) available in the full product. The thin client can be copied to multiple clients running Java SE in support of a WebSphere Application Server Version 8.5 installation.

XSLT 2.0, XPath 2.0, and XQuery 1.0 major new functions

Valuable features have been added to XPath 2.0, XSLT 2.0, and XQuery 1.0 reflecting productivity and feature improvements beyond the XPath 1.0 and XSLT 1.0 standards.

XPath 2.0

- XPath 2.0 has been improved to support the XPath 2.0 and XQuery 1.0 Data Model (XDM), which is based on sequences of heterogeneous items including nodes and primitive types. This replaces and improves on the XPath 1.0 node-set support and becomes the foundation of XSLT 2.0 and XQuery 1.0 data navigation.
- XPath 2.0 adds an extensive collection of functions and operators to allow for an easier programming experience, replacing the XPath 1.0 requirement for proprietary extension mechanisms. These functions and operators help with date and time handling, enhance the string manipulation, support regular expression matching and tokenization, extend the number handling, and add functions for sequence manipulation.
- XPath 2.0 supports schema-aware processing, which allows for data navigation based on XML schema information for not only built-in schema types, but also user-defined schema types.
- XPath 2.0 adds condition (if/then/else branches), iterative (for loops), and quantified expressions (some and every tests) typical of other languages.
- XPath 2.0 adds named collations across multiple functions allowing for locale-specific operation.
- XPath 2.0 provides a backwards-compatibility mode to run most XPath 1.0 expressions unchanged.

XSLT 2.0

- XSLT 2.0 is based on XPath 2.0, allowing XSLT 2.0 to take advantage of all new XPath 2.0 features. Temporary trees have been added to allow navigation of constructed trees during transformation. User-defined functions can be defined in the XSLT language and are callable using XPath 2.0.
- XSLT 2.0 can write to multiple result documents in a single stylesheet execution.
- XSLT 2.0 supports regular expressions to analyze and separate strings.
- XSLT 2.0 allows variables and parameters to be typed, therefore improving the reliability of stylesheets and functions.
- XSLT 2.0 supports schema-aware processing, which allows XSLT 2.0 to check for valid input, temporary trees, and output documents.
- XSLT 2.0 supports initial named templates, which allows the processor to start with a defined template instead of having to match the input document, a feature commonly used with loading documents programmatically using the XPath 2.0 collection and document functions.
- Comparisons in sorting, grouping, and keys are supported with any data type and can use locale-specific named collations.
- XHTML has been added to XSLT 2.0 as a valid output format.
- The next-match instruction allows the same node to be processed with multiple templates.
- The character-map instruction allows fine grained control of serialization of characters.
- XSLT 2.0 added addition instructions for transforming and formatting dates and times.
- XSLT 2.0 added support for tunnel parameters, which allows parameters to be passed through multiple template calls without having to declare the parameter in each template call.
- XSLT 2.0 added multiple mode support to allow templates to apply to specific modes of processing within a stylesheet.
- Unparsed text can be incorporated into the data processed by a stylesheet, which then can be tokenized with the new regular expression support.
- XSLT 2.0 provides a backwards-compatibility mode to run most XSLT 1.0 stylesheets unchanged.

XQuery 1.0

- XQuery 1.0 is based on XPath 2.0, allowing XQuery 1.0 to take advantage of all new XPath 2.0 features. XQuery 1.0 builds on XPath 2.0 to provide full XML Query capability.

- XQuery's FLOWR (For, Let, Order by, Where, Return) expression allows for complicated joins across XML datasets. FLOWR allows for query of large documents or collections of documents. XQuery allows for the mixture of direct XML construction along with computed content returned from FLOWR expressions.
- XQuery has the ability to define functions and variables with syntax that is familiar to users of other languages, allowing larger programs to be defined around the data-query operations.
- XQuery 1.0 supports schema-aware processing, which allows input and constructed documents and elements to be validated.
- XQuery module support allows queries to be broken up into reusable fragments.

Overview of the XML Samples application

The XML Samples application is written to be used with the XML specifications and other documents. However, the most important function that these samples provide is a place to begin experimenting with the XML API and the supported specifications.

Limitations

The XML Samples application is not intended for deployment to production servers. It is for development and educational purposes only. All source code is provided as is for you to use, copy, and modify without royalty payment when you develop applications that run with WebSphere software. You can use the sample code either for your own internal use, for redistribution as part of an application, or in your products.

Content

- The simple API invocation examples included in the samples are intended as simple examples of using the major new features of XPath 2.0, XSLT 2.0, and XQuery 1.0.
  - XPath 2.0 examples
    - Sample 1: Simple XPath invocation

      Shows how to invoke XPath
    - Sample 2: Invoking XPath 1.0 under an XPath 2.0 run time in backwards compatibility mode

      Shows an example that demonstrates differences between XPath 1.0 and XPath 2.0 as well as how to run existing XPath 1.0 statements under XPath 2.0 in backwards-compatibility mode
    - Sample 3: Invoking schema aware XPath 2.0 expressions

      Shows how to run schema-aware expressions; shows how to load schema documents, how to validate input documents, and how to declare namespace prefixes
    - Sample 4: XPath 2.0 - document function (relative URIs) with input and output documents

      Shows how to invoke XPath using the document function with relative URIs
    - Sample 5: XPath running in compiled mode

      Shows how to invoke XPath in compiled mode
    - Sample 6: XPath running in pre-compiled mode

      Shows how to invoke XPath in pre-compiled mode
    - Sample 7: XPath 2.0 collation support

      Shows how to invoke XPath with collation support
  - XSLT 2.0 examples
    - Sample 1: Simple XSLT invocation

      Shows how to invoke XSLT
    - Sample 2: Invoking XSLT 1.0 under an XSLT 2.0 run time in backwards compatibility mode

      Shows differences between XPath 1.0 and XPath 2.0 and how to run existing XSLT 1.0 stylesheets under a XSLT 2.0 processor in backwards-compatibility mode

- Sample 3: XSLT 2.0 updated for-each support

  Shows how to use the XSLT 2.0 for-each functionality
- Sample 4: XSLT 2.0 grouping support

  Shows how to use the capability offered by xsl:for-each-group
- Sample 5: XSLT 2.0 regular expression support

  Shows how to use XSLT 2.0 regular-expression support to work with data in structured legacy formats within XML strings
- Sample 6: XSLT 2.0 date formatting

  Shows how to use XSLT 2.0 date formatting with internationalization
- Sample 7: XSLT 2.0 multiple results

  Shows how to use an XSLT 2.0 result-document instruction to write to multiple outputs simultaneously
- Sample 8: XSLT 2.0 tunnel parameters

  Shows how to use XSLT 2.0 tunnel parameters to allow values to be set and accessible during stylesheet processing
- Sample 9: XSLT 2.0 stylesheet functions

  Shows how to use the XSLT 2.0 stylesheet functions
- Sample 10: XSLT 2.0 initial template

  Shows how to use the XSLT 2.0 initial-template functionality
- Sample 11: XSLT 2.0 template with multiple modes

  Shows how to use the XSLT 2.0 template with multiple modes functionality
- Sample 12: XSLT 2.0 XHTML support - no output method specified

  Shows how to use XSLT 2.0 XHTML support with the XHTML output method
- Sample 13: XSLT 2.0 XHTML support - output method specified

  Shows how to use XSLT 2.0 XHTML support with the XHTML output method
- Sample 14: XSLT 2.0 character maps

  Shows how to use XSLT 2.0 character maps functionality
- Sample 15: XSLT 2.0 "as" attribute

  Shows how to use the XSLT 2.0 "as" attribute functionality
- Sample 16: XSLT 2.0 embedded stylesheets

  Shows how to use the XSLT 2.0 embedded stylesheets functionality
- Sample 17: XSLT 2.0 running in compiled mode

  Shows how to run XSLT in compiled mode
- Sample 18: XSLT 2.0 running in pre-compiled mode

  Shows how to run XSLT in pre-compiled mode
- Sample 19: XSLT 2.0 undeclare-prefixes serialization parameter

  Shows how to use the XSLT undeclare-prefix parameter when producing XML output that is Version 1.1 or higher
- Sample 20: XSLT 2.0 next-match

  Shows how to use the XSLT next-match functionality
- Sample 21: XSLT 2.0 usage of XPath 2.0 collection function

  Shows how to use the collection function
- Sample 22: XSLT 2.0 schema awareness - input validation (valid)

  Shows how to use the stylesheets and schemas to validate input documents
- Sample 23: XSLT 2.0 schema awareness - input validation (invalid)

  Shows how to use the stylesheets and schemas to validate input documents

- Sample 24: XSLT 2.0 schema awareness - temporary tree (valid)
  Shows how to use the validation attribute to validate temporary trees
- Sample 25: XSLT 2.0 schema awareness - temporary tree (invalid)
  Shows how to use the validation attribute to validate temporary trees
- Sample 26: XSLT 2.0 schema awareness - output document (valid)
  Shows how to use the validation attribute to validate the main output document
- Sample 27: XSLT 2.0 schema awareness - output document (invalid)
  Shows how to use the validation attribute to validate the main output document
- Sample 28: XSLT 2.0 schema awareness - element(*, T) function
  Shows how to use the stylesheets and schemas to match on element types instead of names
- Sample 29: XSLT 2.0 use-when
  Shows how to use the use-when functionality
- Sample 30: XSLT 2.0 collation support
  Shows how to use the for-each-group functionality with collations
- Sample 31: Using stylesheet-declared external functions
  Shows how to declare external functions within a stylesheet

– XQuery 1.0 examples
  - Sample 1: Simple XQuery invocation
    Shows how to invoke simple XQuery FLOWR expressions
  - Sample 2: XQuery FLWOR support - using doc function and cross document joins
    Shows how to invoke an XQuery that joins data from multiple documents
  - Sample 3: XQuery declare functions and variables
    Shows how to define and use XQuery functions and variables
  - Sample 4: XQuery TypeDeclaration support
    Shows how to use the TypeDeclaration functionality
  - Sample 5: XQuery running in compiled mode
    Shows how to run XQuery functions in compiled mode
  - Sample 6: XQuery running in pre-compiled mode
    Shows how to invoke XQuery in pre-compiled mode
  - Sample 7: XQuery operations on types (typeswitch, cast as)
    Shows how to use operations on types
  - Sample 8: XQuery schema awareness - input validation (valid)
    Shows how to validate the input document passed to the query
  - Sample 9: XQuery schema awareness - input validation (invalid)
    Shows how to validate the input document passed to the query
  - Sample 10: XQuery schema awareness - node validation (valid)
    Shows how to validate an element using the validate expression
  - Sample 11: XQuery schema awareness - node validation (invalid)
    Shows how to validate an element using the validate expression
  - Sample 12: XQuery schema awareness - element(*, T) function
    Shows how to use schema awareness to match on element types instead of names
  - Sample 13: XQuery modules support
    Shows how commonly used functions and variables can be put in a reusable library module
  - Sample 14: XQuery modules support with schema
    Shows how modules interact with schema support

- Sample 15: Using query-declared external functions

  Shows how to declare external functions within a query

- The Blog Comment Checker examples show how you can search all or your Blogger™ web publishing service blogs for questionable comments. They are examples of high-level applications that use XPath 2.0, XSLT 2.0, and XQuery 1.0.
  - XPath Blog Checker
  - XSLT Blog Checker
  - XQuery Blog Checker
  - Database Integration Checker

Using the XML API to perform operations

You can use the IBM XML Application Programming Interface (API) to perform operations that use the new and updated W3C XML standards.

Before you begin

Limitations of the processor:

- When using the namespace axis, only the namespaces declared on the current node are accessible through the namespace axis (rather than all of the namespaces that are in scope for the current node) if the input document supplied is a StreamSource, SAXSource, or StAXSource.
- When evaluating some string functions and operations, the processor might not handle Unicode characters with code points above #xFFFF correctly; it might incorrectly treat the surrogate pair in the UTF-16 encoding of the character as two separate characters.
- Whitespace text nodes might not be stripped from elements that have complex type with element-only content. When matching children of such an element, use a sequence type that matches only elements such as element() to avoid processing these text nodes.
- The column number reported in an error message relating to an operator expression might not point to the operand that is actually in error. Consider both operands when determining the cause of the error.
- Using a variable declared in an XQuery typeswitch clause as an operand in an arithmetic expression might cause an error.

  When using the variable declared in an XQuery typeswitch clause in an arithmetic expression such as in the following example, the processor might incorrectly report a type error.

  ```
  typeswitch (.)
  case $a as xs:integer return ($a + 1)
  default return 17
  ```

  To work around this limitation, cast the variable to the expected type. For example:

  ```
  typeswitch (.)
  case $a as xs:integer return (($a cast as xs:integer) + 1)
  default return 17
  ```

Procedure

- Perform basic operations.
- Precompile.
- Use resolvers.

- Use external variables and functions.
- Create items and sequences.
- Work with collations.
- Execute using the command-line tools.
- Use a message handler and manage exceptions.

Building and running a sample XML application

You can use the IBM WebSphere Application Server XML thin client, the `com.ibm.xml.thinclient_8.5.0.jar` file, to build a sample XML application. You can also use the API documentation to improve your understanding of the XML API.

Before you begin

1. Install the product.
2. Locate the `com.ibm.xml.thinclient_8.5.0.jar` file.

   You can find the `com.ibm.xml.thinclient_8.5.0.jar` file in your installation tree; for example:

   - *app_server_root*`/runtimes/com.ibm.xml.thinclient_8.5.0.jar`

To see how to build and use an application, refer to the sample application that is packaged with the product.

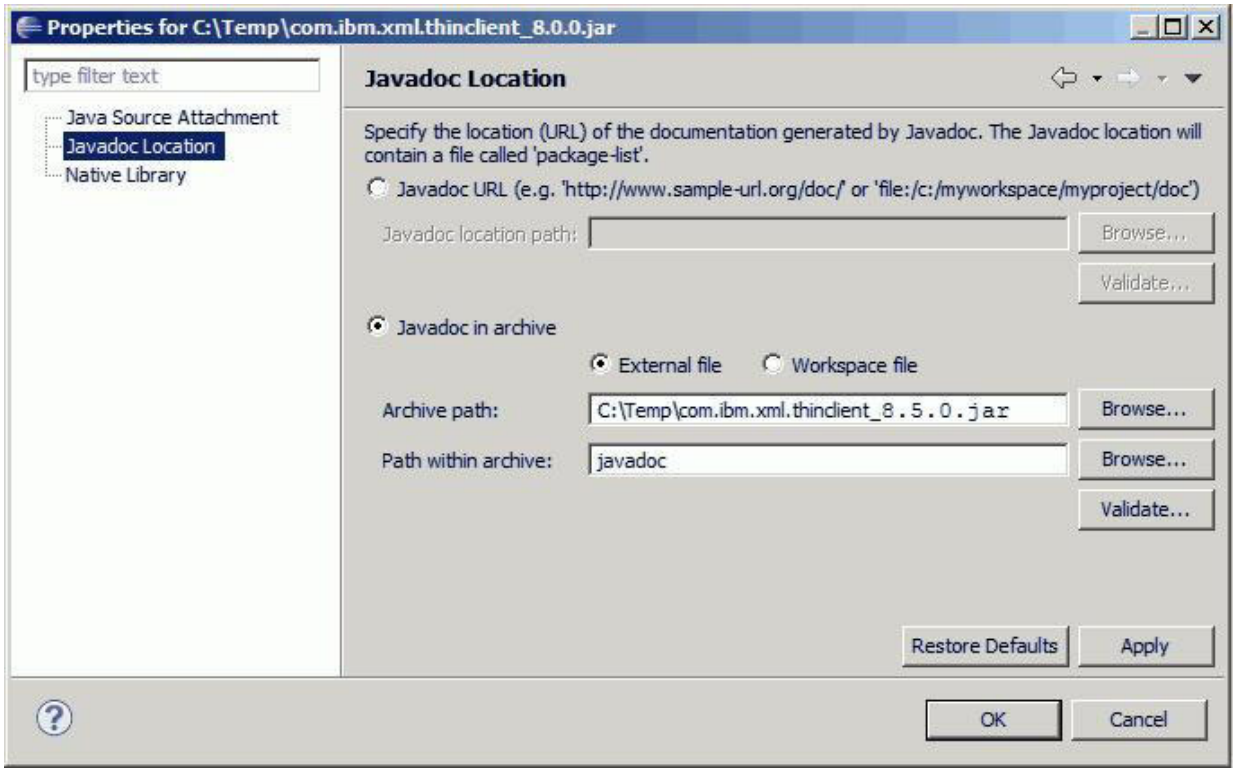
About this task

Follow this procedure when you build and run a sample XML application.

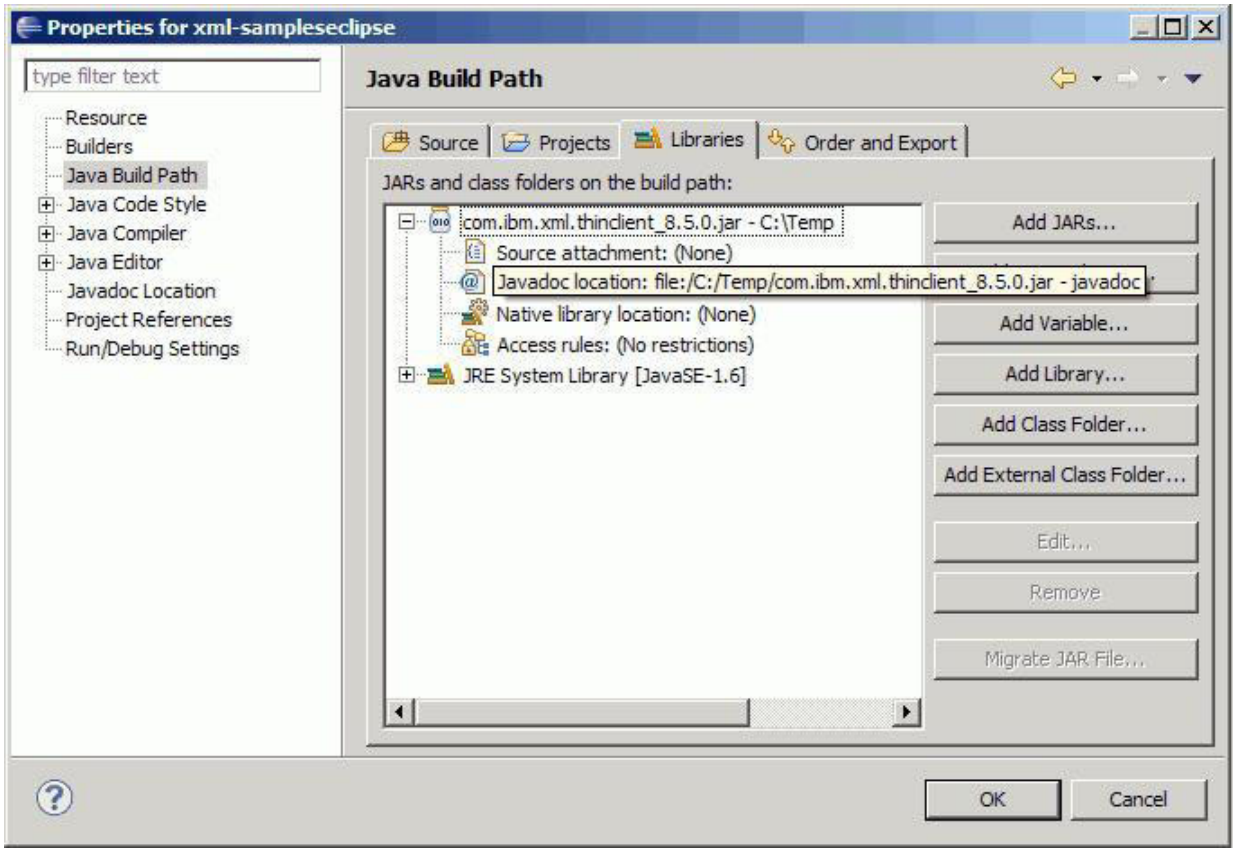
Procedure

1. For build time, include the `com.ibm.xml.thinclient_8.5.0.jar` file in the build-time class path while developing your sample XML application.

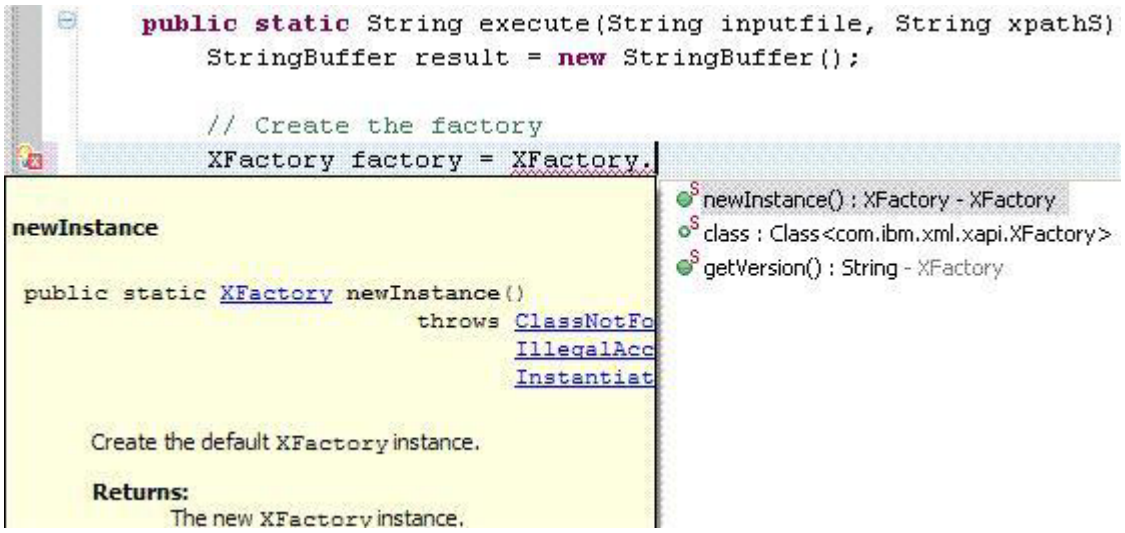
   Also, attach the API documentation from the `javadoc` directory that is inside the `com.ibm.xml.thinclient_8.5.0.jar` file.



The results of these actions are shown in the following image:



When complete, your application should compile; and when using context completion, you should have access to the API documentation as shown here:



2. Deploy your application.

Running the IBM Thin Client for XML

The IBM Thin Client for XML with WebSphere Application Server allows applications to take advantage of IBM XML technology components in a simple Java environment. Such applications can directly access the XML API to process XPath, XQuery, and XSL in a Java SE runtime environment. You can bundle the Thin Client for XML library with your application by using the WebSphere Application Server installation. The Thin Client for XML also extends the choice of Java SE runtime environment. It can be run in IBM Java Runtime Environments (JREs) packaged with WebSphere Application Server as well as in the Windows and Linux JREs from Oracle Corporation, the JREs that are downloaded from the Oracle Corporation website, or the JRE that is downloaded from the HP website.

Before you begin

Before you can use the API provided by the Thin Client for XML you must obtain the Thin Client for XML Java archive (JAR) file. This can be accomplished by installing WebSphere Application Server. The Thin Client for XML JAR file, `com.ibm.xml.thinclient_8.5.0.jar`, is located in the *app_server_root*`\runtimes` directory.

Copy the JAR file, `com.ibm.xml.thinclient_8.5.0.jar`, to other machines to enable access to the XML API. Copies of the Thin Client for XML are subject to the terms and conditions of the license agreement for the WebSphere Application Server Version 8.5 product from which you obtained the Thin Client for XML. Refer to the license agreements for correct usage and other limitations.

The Thin Client for XML with WebSphere Application Server runs on distributed operating systems with Java SE Development Kit (JDK) support, including Java 6.0 and Java 7.0.

About this task

Run the Thin Client for XML by completing this procedure.

Procedure

Invoke the client application.
Run the following Java command:

```
java_install_root/bin/java
-classpath com.ibm.xml.thinclient_8.5.0.jar:list_of_your_application_jars_and_classes
fully_qualified_class_name_to_run
```

Performing basic operations

You can use this information to help you to perform basic XPath, XSLT, and XQuery operations.

Procedure

- Perform basic XPath operations.
- Perform basic XSLT operations.
- Perform basic XQuery operations.
- View the results.
- Serialize the results.
- Navigate with XSequenceCursor.
- Navigate with XTreeCursor.

Overview of the XML API

The product provides the IBM XML Application Programming Interface in support of the XML standards. This application programming interface invokes a runtime engine that is capable of executing XPath 2.0, XSLT 2.0, and XQuery 1.0 as well as manipulating the returned XML data.

Supported input and result types:

The API supports standard implementations of Java API for XML Processing (JAXP) input and result types.

The following standard implementations of the JAXP Source interface are supported as the input to the execute methods of an executable object (XPathExecutable, XQueryExecutable, and XSLTExecutable) as well as to other methods that take a Source object such as XItemFactory.item(Source), XDynamicContext.bind(QName, Source), and so on. They are also accepted for the stylesheet parameter to the XFactory.prepareXSLT methods. These classes are all included in the Java 6.0 and Java 7.0 versions of the JRE. Refer to the Java API documentation for more information.

- javax.xml.transform.stream.StreamSource
- javax.xml.transform.dom.DOMSource
- javax.xml.transform.sax.SAXSource
- javax.xml.transform.stax.StAXSource

The XSourceResolver.getSource method can return any of the previous implementations of source as well as a further implementation, XItemSource, provided by the API. This allows results from a previous query or transformation to be used as input to a new query or transformation without first serializing to a DOM or stream. The XItemSource can simply be initialized with the XItemView instance and returned by the XSourceResolver implementation.

The following standard implementations of the JAXP Result interface are supported for those execute methods that take a result as well as other methods that take a result object such as the export methods on XItemView and XSequenceCursor. These classes are included in the Java 6.0 and Java 7.0 versions of the JRE. Refer to the Java API documentation for more information.

- javax.xml.transform.stream.StreamResult
- javax.xml.transform.dom.DOMResult
- javax.xml.transform.sax.SAXResult
- javax.xml.transform.stax.StAXResult

A further implementation of the Result interface, XSequenceCursorResult, is provided by the API to allow a result to be returned as an XSequenceCursor. The execute methods on the executable objects that do not take a result object return the result as an XSequenceCursor already; therefore, this is primarily for use with XSLT multiple result documents. An implementation of XResultResolver can return an

XSequenceCursorResult to the processor and the processor will populate the XSequenceCursorResult with the XSequenceCursor and the associated output parameters during processing. After the transformation is complete the XSequenceCursor can be retrieved by the application by calling the getSequenceCursor method on the XSequenceCursorResult object. The output parameters can be retrieved by calling the getOutputParameters method. The output parameters then can be used on a call to the XSequenceCursor exportSequence method.

The XFactory class:

This is a quick overview of the methods that the XFactory class provides. The individual methods are discussed in detail in other articles in the information center.

The XFactory class is the main factory class for creating executables for XPath, XQuery, and XSLT. It is also the means for creating instances of other classes and factories such as the XStaticContext, XDynamicContext, XItemFactory, and XSequenceTypeFactory classes. An instance of XFactory maintains a set of registered schemas as well and can be validating or non-validating. A validating factory produces schema-aware executables and ensures that source documents get validated against the set of registered schemas before they are processed. If different sets of stylesheets or expressions need different sets of schemas, these can be kept separate by using more than one XFactory instance. An XFactory instance can be created by calling the static newInstance() method on the XFactory class. The instance is thread safe as long as the settings remain stable.

Here is an example of using the XFactory class.

```
// Create a new XFactory
XFactory factory = XFactory.newInstance();

// Create an XPath executable
XPathExecutable executable = factory.prepareXPath("/something/bar[2]");

// Create a new XStaticContext
XStaticContext staticContext = factory.newStaticContext();

// Create an XPath executable that is compiled in backwards
// compatibility mode
staticContext.setXPathCompatibilityMode(XStaticContext.XPATH1_0_BC_COMPATIBILITY);
XPathExecutable bcExecutable = factory.prepareXPath("/something/bar[2]", staticContext);

// Set validating
factory.setValidating(true);

// Register a schema
factory.registerSchema(new StreamSource("myschema.xsd"));

// Create a schema aware XPath executable
staticContext = factory.newStaticContext();
staticContext.declareNamespace("something", "http://myschema/something");
XPathExecutable schemaExecutable =
   factory.prepareXPath("/something:something/bar[2] instance of element(bar, something:barType)", staticContext);

// Get the item factory for this XFactory (the two are related
// since the item factory depends on the schemas that are
// registered with XFactory).
XItemFactory itemFactory = factory.getItemFactory();
```

Relationship of the processor to the Java API for XML Processing (JAXP):

In most cases, you will need to migrate any of your applications that used the Java API for XML Processing (JAXP) to use the current API.

The XSLT and XPath processing portions of JAXP are defined with reference to XSLT 1.0 and XPath 1.0. JAXP does not have provisions for XSLT 2.0 and XPath 2.0 processing. In particular, there are situations in which an XSLT 2.0 or an XPath 2.0 processor must produce a result that is different from that produced by an XSLT 1.0 or an XPath 1.0 processor given identical input and an identical stylesheet or expression. Therefore, it is not possible to instantiate the current processor using JAXP.

JAXP also does not have support for sequences, XQuery 1.0, or the many current data types that are available in XSLT 2.0, XPath 2.0, and XQuery 1.0. JAXP also is limited in the forms that input and output can take. All these things make it a poor fit for processing XSLT 2.0 stylesheets as well as XPath 2.0 and XQuery 1.0 expressions.

The following examples demonstrate common migration scenarios and how to write code using the current API that is equivalent to code that you might have written using JAXP.

Processing an XSLT stylesheet using the API

The following example demonstrates how to process an XSLT stylesheet and apply it to some input to produce an instance of the javax.xml.transform.Result interface.

```
XFactory factory = XFactory.newInstance();
XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
style.execute(new StreamSource("input.xml"), new StreamResult(System.out));
```

Processing an XPath expression using the API

The following example demonstrates how to process an XPath expression and apply it to some input.

```
XFactory factory = XFactory.newInstance();
XPathExecutable pathExpr = factory.prepareXPath("/doc/child[@id='N1378']");

// Process input from a StreamSource
XSequenceCursor result1 = pathExpr.execute(new StreamSource("input.xml"));

// Process input from a DOM node
XSequenceCursor result2 = pathExpr.execute(new DOMSource(node));
```

Resolving URI references

If you used instances of the JAXP URIResolver interface to resolve references to the XSLT document() function, you can now use the XSourceResolver interface to accomplish the same thing. To resolve references to the document() function or the fn:doc() function, you can set an instance of the XSourceResolver interface on an instance of the XDynamicContext interface; to resolve references to stylesheets imported through xsl:import and xsl:include declarations, you can set an instance of the XSourceResolver interface on an instance of the XStaticContext interface.

The following example shows how you might set up an instance of the XSourceResolver interface that treats input documents as XSLT stylesheets and uses them to generate the input data for reference to the doc or document functions in another XSLT stylesheet.

```
final XFactory factory = XFactory.newInstance();
XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
XDynamicContext dContext = factory.newDynamicContext();

// Create and set an instance of an anonymous inner class as the
// XSourceResolver
dContext.setSourceResolver(new XSourceResolver() {
    // Create an item to use as the initial context node for
    // transformations in the getSource method
    private XItemView fDummyNode =
        factory.getItemFactory()
            .item(new StreamSource(
                new StringReader("<doc/>")));

    // Resolve URIs by loading the resource as an XSLT stylesheet
    // and evaluating it - return the result as the Source to use
    public Source getSource(String href, String base) {
        java.net.URI baseURI;
        try {
            // Get base URI object
            baseURI = new java.net.URI(base);
        } catch (java.net.URISyntaxException use) {
            throw new RuntimeException(use);
        }
        // Resolved relative reference against base URI
        String resolvedURI = baseURI.resolve(href).toString();

        // Prepare and execute the stylesheet
        XItemView transformResult =
            factory.prepareXSLT(new StreamSource(resolvedURI))
                .execute(fDummyNode);
```

```
        return new XItemSource(transformResult);
    }
});

XSequenceCursor result = style.execute(new StreamSource("input.xml"), dContext);
```

Defining extension functions and external functions

Using JAXP for XPath expression evaluation, you could register an instance of the XPathFunctionResolver interface to supply the implementations of extension functions that your XPath expressions might call. The XSLT portion of JAXP does not have an equivalent mechanism.

With the current API, you can declare extension functions on an instance of the XStaticContext interface, specifying the expected types of the arguments and the expected type of the result of calling the function, and you can register the implementations of extension functions on an instance of the XDynamicContext interface. Your XSLT stylesheet and XPath and XQuery expressions can call any extension functions that you register.

Setting the values of stylesheet parameters and external variables

Using JAXP, you could supply the initial values of stylesheet parameters by calling the Transformer.setParameter method and you could supply the values of variables for XPath expressions by supplying an instance of the XPathVariableResolver interface. Using the API, you can declare variables using the declareVariable() methods of the XStaticContext interface, specifying a variable name and the expected type of the variable. You can supply the values of stylesheet parameters, XPath variables, and XQuery external variables through one of the bind() methods of the XDynamicContext interface.

The following example shows how you might use a variable in an XPath expression to look up product entries in a catalog based on the product identifier.

```
XFactory factory = XFactory.newInstance();
XStaticContext sContext = factory.newStaticContext();

// Declare the XPath variable "query-id" in the static context
QName queryIdVar = new QName("query-id");
sContext.declareVariable(queryIdVar, XTypeConstants.STRING_QNAME);

// Prepare the XPath expression
XItemFactory itemFactory = factory.getItemFactory();
XPathExecutable expr =
    factory.prepareXPath("/catalog/product[id eq $query-id]", sContext);

XItemView catalog = itemFactory.item(new StreamSource("catalog.xml"));
XDynamicContext dContext = factory.newDynamicContext();

// Set the value of the "query-id" variable, and evaluate the
// expression with that variable value
dContext.bind(queryIdVar, "ID43785");
XSequenceCursor product1 = expr.execute(catalog, dContext);

// Set the value of the "query-id" variable, and evaluate the
// expression with the new variable value
dContext.bind(queryIdVar, "ID18574");
XSequenceCursor product2 = expr.execute(catalog, dContext);
```

Identity transformation

Another operation that is frequently used in JAXP is the identity transformation. This is a convenient way of transforming data from one form to another—for example, serializing a DOM tree, or producing a DOM tree from SAX events. It is possible to perform identity transformations using the API. See “Performing basic XSLT operations” on page 1863 for an example.

Prepare-time and execution-time configuration

In JAXP you supply much of the runtime configuration information for XSLT stylesheets – the values of stylesheet parameters, URIResolvers, and so on—directly on the objects that are used to perform transformations – instances of the Transformer interface and the TransformerHandler interface. Similarly,

you supply configuration information for the preparation of stylesheets and XPath expressions directly on instances of the TransformerFactory and XPathFactory classes in JAXP.

With the API, you can supply configuration information that is needed at the time a stylesheet or an expression is prepared—namespace bindings, the types of external functions or variables, and so on—using an instance of the XStaticContext interface. Similarly, you can provide any configuration information that is needed to evaluate a stylesheet or expression—the values of variables, settings of output parameters, and so on—on an instance of the XDynamicContext interface, which you can pass as an argument to the execute methods of the XExecutable interface and its subinterfaces.

This separation of the configuration information into a separate object makes the API more thread safe. Your application can use the same instance of the XExecutable interface on different threads without any synchronization. This stands in contrast to JAXP, where instances of the Transformer, TransformerHandler and XPathExpression interfaces are not thread safe; every thread that uses them has to synchronize access to shared instances of those objects or create distinct copies that are specific to each thread.

Handling errors

In JAXP, you could supply an instance of the ErrorHandler interface to control how the processor should respond to errors. In the API, you can achieve this by supplying an instance of the XMessageHandler interface on an instance of the XStaticContext interface for preparation-time errors or the XDynamicContext interface for execution-time errors.

Performance tips:

Follow these tips to improve performance when using XPath, XQuery, and XSLT.

Table 251. Performance tips. Follow these tips to improve performance.

Language	Tip
XPath	Using // can be an expensive operation. Making the path more explicit (a/b/c rather than a//c) can improve performance. This is especially important when the path starts at or near the root of a large document.
	The last() function, because it requires fully evaluating the sequence to count the items, can be an expensive operation.
	Positional predicates with constant values, such as [3], are usually more efficient than those with values that are computed or that are retrieved from variables.
XQuery	Consider calling the registerImportedSchemas method on the XQueryExecutable instance if the query imports one or more schemas and the same query will be executed more than once. If registerImportedSchemas is not called, the imported schemas will be loaded every time one of the execute methods is called. By default, imported schemas can only be used to validate result trees; but calling this method has the same effect as registering the schemas with the XFactory, which means that they will be used to validate input documents.

Table 251. Performance tips (continued). Follow these tips to improve performance.

Language	Tip
XSLT	Parameters are slower to access than variables. If you do not need to supply the value of the parameter externally, use a variable.
	Using xsl:key elements and the key() function can be an efficient way to retrieve node sets.
	Pattern matching and apply-templates dispatch are usually faster than the xsl:if or xsl:when statements.
	Positional predicates in match patterns are usually expensive.
	In general, simpler match patterns such as "address" are less expensive to process than complicated ones such as "/purchaseorder/shipping/customer/postal/address". Take advantage of your knowledge of the document's structure and your stylesheet's behavior to avoid unnecessarily overspecifying.
	For some data models, the xsl:skip-space operation must be applied during document navigation rather than during document load. This can add some execution-time overhead.
	Consider calling the registerImportedSchemas method on the XSLTExecutable instance if the stylesheet imports one or more schemas and the same stylesheet will be executed more than once. If registerImportedSchemas is not called, the imported schemas will be loaded every time one of the execute methods is called. By default, imported schemas can only be used to validate result trees; but calling this method has the same effect as registering the schemas with the XFactory, which means that they will be used to validate input documents.
XPath, XQuery, and XSLT	Decoding and encoding is expensive. Generally, UTF-8 and UTF-16 can be read and written more quickly than other encodings.

XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-specific behaviors:

Table 252. Implementation-defined behaviors.

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
multi preserve-strip-space matched	It is a recoverable dynamic error if an element in the source document matches both an xsl:strip-space and an xsl:preserve-space declaration.	XSLT 2.0 Section 4.4	XTRE0270	The processor does not return an error and recovers by selecting the xsl:strip-space or xsl:preserve-space declaration that occurs last in declaration order.
pattern evaluation	Any dynamic error or type error that occurs during the evaluation of a pattern against a particular node is treated as a recoverable error even if the error would not be recoverable under other circumstances. The optional recovery action is to treat the pattern as not matching that node.	XSLT 2.0 Section 5.5.4		The processor does not return an error and recovers by treating the pattern as not matching that node.
multiple templates matched	It is a recoverable dynamic error if the conflict resolution algorithm for template rules leaves more than one matching template rule. The optional recovery action is to select, from the matching template rules that are left, the one that occurs last in declaration order.	XSLT 2.0 Section 6.4	XTRE0540	The processor does not return an error and recovers by selecting the template that occurs last in declaration order.
invalid value-xml space	It is a recoverable dynamic error if the name of a constructed attribute is xml:space and the value is not either default or preserve. The optional recovery action is to construct the attribute with the value as requested. This applies whether the attribute is constructed using a literal result element or by using the xsl:attribute, xsl:copy, or xsl:copy-of instructions.	XSLT 2.0 Section 11.1.2	XTRE0795	The processor does not return an error and constructs the attribute with the value as requested.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
invalid fragment in document uri	When a URI reference supplied to the document function contains a fragment identifier, it is a recoverable dynamic error if one of the following is true: • the media type is not one that is recognized by the processor • the fragment identifier does not conform to the rules for fragment identifiers for that media type • if the fragment identifier selects something other than a sequence of nodes (if it selects a range of characters within a text node, for example) The optional recovery action is to ignore the fragment identifier and return the document node. (Note that the recovery option is different from that in XSLT 1.0)	XSLT 2.0 Section 16.1	XTRE1160	The processor ignores the fragment identifier and returns the document node. No warning is returned.
same resource for multiple results	It is a recoverable dynamic error for a transformation to generate two or more final result trees with URIs that identify the same physical resource. The optional recovery action is implementation-dependent because it may be impossible for the processor to detect the error.	XSLT 2.0 Section 19.1	XTRE1495	The processor returns an error.
read and write to same resource	It is a recoverable dynamic error for a stylesheet to write to an external resource and read from the same resource during a single transformation, whether or not the same URI is used to access the resource in both cases. The optional recovery action is implementation-dependent; implementations are not required to detect the error condition.	XSLT 2.0 Section 19.1	XTRE1500	The processor does not attempt to detect this error condition.
disable output escaping not supported	It is a recoverable dynamic error if an xsl:value-of or xsl:text instruction specifies that output escaping is to be disabled and the implementation does not support this. The optional recovery action is to ignore the disable-output-escaping attribute.	XSLT 2.0 Section 20.2	XTRE1620	The processor does not provide the ability to disable output escaping and ignores the disable-output-escaping attribute without warning.
disable output escaping for not serialized	It is a recoverable dynamic error if an xsl:value-of or xsl:text instruction specifies that output escaping is to be disabled when writing to a final result tree that is not being serialized. The optional recovery action is to ignore the disable-output-escaping attribute.	XSLT 2.0 Section 20.2	XTRE1630	The processor does not provide the ability to disable output escaping and ignores the disable-output-escaping attribute without warning.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
integer overflow	On overflow and underflow situations during xs:integer arithmetic operations, implementations that support limited-precision integer operations must select from the following options: • Always raising an error (FOAR0002) • Providing an implementation-defined mechanism that allows users to choose between raising an error and returning a result that is modulo the largest representable integer value	XQuery 1.0 and XPath 2.0 Functions and Operators Section 6.2	FOAR0002	The processor provides a mechanism to enable overflow detection for integer operations through the `com.ibm.xml.xapi.XStaticContext.setIntegerMathMode(int)` method, which can be called with the constants `INTEGER_MATH_MODE_OVERFLOW_DETECTION` or `INTEGER_MATH_MODE_LIMITED_PRECISION` (defined on the XStaticContext interface) to choose between the two options. A third option, `INTEGER_MATH_MODE_ARBITRARY_PRECISION`, is provided to allow arbitrary precision integers.
double float overflow	For xs:float and xs:double arithmetic operations, overflow behavior must conform to IEEE 754-1985, which allows the following options: • Raising an error (FOAR0002) using an overflow trap • Returning INF or -INF • Returning the largest (positive or negative) non-infinite number	XQuery 1.0 and XPath 2.0 Functions and Operators Section 6.2		The processor returns INF or -INF.
double float underflow	For xs:float and xs:double arithmetic operations, underflow behavior must conform to IEEE 754-1985, which allows the following options: • Raising an error (FOAR0002) using an underflow trap • Returning INF or -INF • Returning 0.0E0, +/- 2**Emin, or a denormalized value where Emin is the smallest possible xs:float or xs:double exponent	XQuery 1.0 and XPath 2.0 Functions and Operators Section 6.2		The processor returns 0.0E0.
invalid use of doctype-system or standalone attr	It is a serialization error to specify the doctype-system parameter or to specify the standalone parameter with a value other than omit if the instance of the data model contains text nodes or multiple element nodes as children of the root node. The serializer must either signal the error or recover by ignoring the request to output a document type declaration or standalone parameter.	SR Section 5	SEPM0004	The processor returns a serialization error.
invalid xml output encoding	A serialization error occurs if the output method is XML and an output encoding other than UTF-8 or UTF-16 is requested and the serializer does not support that encoding. The serializer must return the error (SESU0007) or recover by using UTF-8 or UTF-16 instead.	SR Section 5.1.2	SESU0007	The processor returns a serialization error.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
error or recovery	Some dynamic errors are classed as recoverable errors. When a recoverable error occurs, the processor may signal the error (by reporting the error condition and terminating execution) or take a defined recovery action and continue processing.	XSLT 2.0 Section 2.9		The processor recovers from many of the recoverable errors but signals an error in some cases. Consult this table to determine the implementation-defined behavior of each type of recoverable error.
signal type errors statically	Whether or not type errors are signaled statically is implementation defined.	XSLT 2.0 Section 2.9		The processor detects type errors statically whenever possible; but there are cases where the error cannot be detected at compile time. In these cases, the error is detected at run time.
handling serialization errors	The handling of serialization errors is implementation defined.	XSLT 2.0 Section 20		Error messages are sent to a `com.ibm.xml.xapi.XMessageHandler` implementation that has been registered using the `setXMessageHandler` method on the `com.ibm.xml.xapi.XDynamicContext` interface.
base output URI	The way in which a base output URI is established is implementation defined.	XSLT 2.0 Section 2.3⌂		Users can set the base output URI using the `setBaseOutputURI` method on the `com.ibm.xml.xapi.XDynamicContext` interface.
extension attributes	Implementations may allow extension attributes to modify the behavior of extension functions and extension instructions or to influence the behavior of the serialization methods xml, xhtml, html, or text to the extent that the behavior of the serialization method is implementation defined or implementation-dependent.	XSLT 2.0 Section 3.3		The processor recognizes the indent-amount attribute on xsl:output declarations to specify how many spaces are used for each indentation level when indentation is enabled. The attribute must be from one of the following namespaces: • http://www.ibm.com/xmlns/prod/xltxe-j • http://xml.apache.org/xalan • http://xml.apache.org/xslt See the additional serialization params item later in the table.
user-defined data elements	An implementation may attach an implementation-defined meaning to user-defined data elements that appear in particular namespaces. The set of namespaces that are recognized for such data elements is implementation defined.	XSLT 2.0 Section 3.6.2		The processor does not recognize any user-defined data elements.
user-defined types	Support for additional user-defined or implementation-defined types is implementation defined.	XPath 2.0 Data Model Section 2.6		A schema-aware processor is supported.
undefined type behavior	Some typed values in the data model are undefined. Attempting to access an undefined property is always an error. Behavior in these cases is implementation defined and the host language is responsible for determining the result.	XPath 2.0 Data Model Section 2.5.2, Bullet 4.d	FOTY0012	The processor returned an error. Error messages are sent to a `com.ibm.xml.xapi.XMessageHandler` implementation that has been registered using the `setXMessageHandler` method on the `com.ibm.xml.xapi.XDynamicContext` interface.
namespace node representation	Representation of namespaces, that is whether or not they are represented as nodes, is implementation dependent.	XPath 2.0 Data Model Section 6.4		The namespace axis is supported.
locate stylesheet module	After resolving against the base URI, the way in which the URI reference from the href attribute of an xsl:include or xsl:import declaration is used to locate a representation of a stylesheet module and the way in which the stylesheet module is constructed from that representation are implementation defined. In particular, which URI schemes are supported, whether fragment identifiers are supported, and what media types are supported are implementation defined.	XSLT 2.0 Section 3.10.1		Users may provide an `com.ibm.xml.xapi.XSourceResolver` implementation through the `com.ibm.xml.xapi.XStaticContext.setSourceResolver(XSourceResolver)` method. The `XSourceResolver` is used by the processor to resolve URIs from xsl:include and xsl:import declarations (therefore, the user may decide which URIs to support), and its `getSource(String, String)` method must return a JAXP Source object. If no `XSourceResolver` is given, the processor handles the file URI scheme and those supported by the `java.net.URL.openConnection()` method. A URI fragment may be used to select an embedded stylesheet module within a source XML document. The fragment must identify an xsl:stylesheet element in the document by using the value of one of its attributes that is an xml:id attribute or is defined in a DTD as being of type ID or is defined in a schema as being of type xs:ID. There is no built-in support for non-XML media types, but users may use an `XSourceResolver` implementation to provide the processor with an XML representation of non-XML data.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
extension functions	Extension functions The XSLT 2.0 specification defines how extension instructions and extension functions are invoked, but the facilities for creating new extension instructions and extension functions are implementation defined.	XSLT 2.0 Section 18.1		Extension functions are supported through the processor's `XStaticContext.declareFunction` and `XDynamicContext.bindFunction` methods. Alternatively, it is also possible to declare extension functions directly within a stylesheet or query. The old style of extension functions supported in the parser and the processor are also supported for backwards compatibility; however, only 1.0 types are supported with the old style. See the API documentation for more information. It is recommended that you declare extension functions directly in your stylesheet or through the API for extension functions whenever possible rather than use the old style.
extension instructions	Extension instructions The XSLT 2.0 specification defines how extension instructions and extension functions are invoked, but the facilities for creating new extension instructions and extension functions are implementation defined.	XSLT 2.0 Section 18.2	XTDE1450	User-defined extension instructions are not supported. Extension instructions must be protected with a fallback instruction; otherwise, the stylesheet will fail to compile. The redirect extension is supported for backwards compatibility; however, it is recommended that you use the XSLT 2.0 xsl:result-document instruction because this is more portable.
backwards-compatiblility	Whether a particular XSLT 2.0 implementation supports backwards-compatible behavior is implementation defined.	XSLT 2.0 Section 3.8		The backwards-compatibility feature as described in the XSLT 2.0 specification is supported.
in-scope collations for use-when	In-scope collations for use-when expressions	XSLT 2.0 Section 3.12		The only collation available during the evaluation of a use-when is the Unicode code-point collation.
current date time for use-when	Current date and time for use-when expressions	XSLT 2.0 Section 3.12		The current date time is the current date time of the system as retrieved by calling the `java.util.GregorianCalendar getInstance()` method.
implicit timezone for use-when	Implicit time zone for use-when expressions	XSLT 2.0 Section 3.12		The implicit time zone is the time zone of the system as retrieved by calling the `java.util.TimeZone getDefault()` method.
maximum number of decimal digits	Maximum number of total digits in decimal digits is implementation defined , but they must be at least 18 digits.	XSLT 2.0 Section 4.6		The implementation uses the `java.math.BigDecimal` class, which supports nearly unlimited precision, except that the number of digits to the right of the decimal place is limited to Integer.MAX_VALUE. Truncation is only required in the case of division, where there is the possibility of non-terminating decimals. The precision of the fractional part of the result is limited to 18 digits. The rounding mode in this case is ROUND_HALF_UP, where discarded fractions of 0.5 or later are rounded up (away from zero) and lesser fractions are rounded down.
year component values	For the xs:date, xs:time, xs:dateTime, xs:gYear, and xs:gYearMonth types: the range of values of the year component (which must be at least +0001 to +9999), and the maximum number of fractional second digits (which must be at least 3)	XSLT 2.0 Section 4.6		The year component of the xs:date, xs:dateTime, xs:gYear, and xs:gYearMonth types has the range -(10^9-1) to (10^9)-1. The maximum number of fractional seconds digits supported for the xs:time and xs:dateTime types is three.
duration	For the xs:duration type: the maximum absolute values of the years, months, days, hours, minutes, and seconds components	XSLT 2.0 Section 4.6		The maximum absolute values for the duration components are: • year: 178956970 • month: 2147483647 • day: 106751991167 • hour: 2562047788015 • minute: 153722867280912 • second: 9223372036854775 Three digits of precision are supported for milliseconds.
year month duration	For the xdt:yearMonthDuration type: the maximum absolute value expressed as an integer number of months	XSLT 2.0 Section 4.6		The maximum absolute value for the xs:yearMonthDuration type is 2147483647 months.
day time duration	For the xdt:dayTimeDuration type: the maximum absolute value expressed as a decimal number of seconds	XSLT 2.0 Section 4.6		The maximum absolute value for the xs:dayTimeDuration type is 9223372036854775 seconds.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
value maximum length	For the xs:string, xs:hexBinary, xs:base64Binary, xs:QName, xs:anyURI, xs:NOTATION types and types derived from them: the maximum length of the value	XSLT 2.0 Section 4.6		The theoretical maximum length is 2^31 - 1; however, the system is likely to run out of memory long before that limit is reached.
sequence length	Maximum number of items in a sequence	XSLT 2.0 Section 4.6		The theoretical maximum number of items is 2^31 - 1; however, the system is likely to run out of memory long before that limit is reached.
statically known collations	Set of in-scope collations	XSLT 2.0 Section 5.4.1 XPath 2.0 Section 2.1.1 XQuery 1.0 Section 2.1.1		Any URI is considered to be in the set of in-scope collations. Every URI that is used as a collation URI must be associated with a Java Collator at execution time.
implicit timezone	Implicit time zone	XSLT 2.0 Section 5.4.3.2 XPath 2.0 Section 2.1.2 XQuery 1.0 Section 2.1.2		The implicit time zone can be set using the `XDynamicContext.setImplicitTimeZone(Duration)` method in the processor. If the implicit time zone is not set, the system time zone is used as retrieved through the `java.util.TimeZone.getDefault()` method.
default collection	Default collection This is the sequence of nodes that would result from calling the fn:collection function with no arguments. The value of the default collection may be initialized by the implementation.	XSLT 2.0 XPath 2.0 Section 2.1.2	FODC0004	The default collection is determined by the XCollectionResolver registered with the XDynamicContext. If no XCollectionResolver is registered an error is returned and the empty sequence is used. For more information about the XCollectionResolver interface, refer to the API documentation.
stylesheet parameter	Supply stylesheet parameter value when executing a transformation A top-level xsl:param element declares a stylesheet parameter. A stylesheet parameter is a global variable with the additional property that its value can be supplied by the caller when a transformation is initiated.	XSLT 2.0 Section 9.5		If a value for a parameter is bound in the XDynamicContext, that value is used; otherwise, the default value specified in the stylesheet for the parameter is used. To bind values to parameters, use the `XDynamicContext.bind()` methods.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
stylesheet function override mechanism	Override a stylesheet function to a function provided by the implementer; pertains to the xsl:function element The optional override attribute defines what happens if this function has the same name and arity as a function provided by the implementer or made available in the static context using an implementation-defined mechanism. If the override attribute has the value yes, this function is used in preference; if it has the value no, the other function is used in preference. The default value is yes. Specifying override="yes" ensures interoperable behavior—the same code will execute with all processors. Specifying override="no" is useful when writing a fallback implementation of a function that is available with some processors but not others—it allows the vendor's implementation of the function (or a user's implementation written as an extension function) to be used in preference to the stylesheet implementation.	XSLT 2.0 Section 10.3		If the override attribute had the value of yes, stylesheet functions can override the following: • extension functions declared in the `XStaticContext` using the `XStaticContext.declareFunction()` methods and bound in the `XDynamicContext` using the `XDynamicContext.bindFunction()` methods • old style extension functions supported in XSLT4J and XL TXE-J 1.0 See Using extension functions. • EXSLT extension functions implemented by the processor Stylesheet functions cannot be made to override any of the core functions defined in the XSLT 2.0 specification or the XQuery 1.0 and XPath 2.0 Functions and Operators specification.
normalize copied xml id	When an xml:id attribute is copied using either the xsl:copy or xsl:copy-of instruction, it is implementation defined whether the value of the attribute is subjected to attribute value normalization—that is, effectively applying the normalize-space XQuery 1.0 and XPath 2.0 Functions and Operators function.	XSLT 2.0 Section 11.9		Neither the xsl:copy and xsl:copy-of instructions apply the normalize-space function on xml:id attributes; so all white spaces are preserved.
numbering sequences supported	Numbering sequences supported Which numbering sequences, additional to those listed previously, are supported is implementation defined. If an implementation does not support a numbering sequence represented by the given token, it must use a format token of 1.	XSLT 2.0 Section 12.3		The processor only supports the standard sets of numbering sequences.
bounds on range of numbers	Lower and upper bounds on the range of numbers For the standard numbering sequences, any upper bound imposed by the implementation must not be less than 1000 (one thousand) and any lower bound must not be greater than 1. Numbers that fall outside this range must be formatted using the format token 1. The numbering sequence associated with the format token 1 has a lower bound of 0 (zero).	XSLT 2.0 Section 12.3		• Alphabet values: arbitrary precision when in integer arbitrary precision mode, otherwise 0 to 2^63 - 1 • Digit values: arbitrary precision when in integer arbitrary precision mode, otherwise 0 to 2^63 - 1 • Roman values: 0 - 9999 • Word number: 0 - 19999

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
default language for numbering	Default language for numbering The lang attribute specifies which language's conventions are to be used; it has the same range of values as xml:lang (see the XML 1.0 specification). If no lang value is specified, the language that is used is implementation defined. The set of languages for which numbering is supported is implementation defined. If a language is requested that is not supported, the processor uses the language that it would use if the lang attribute were omitted.	XSLT 2.0 Section 12.3		The default language for number formatting is always set to en-US.
languages for numbering	Set of languages for numbering Many numbering sequences are language sensitive.	XSLT 2.0 Section 12.3		The languages supported are based on International Component Unicode (ICU) and languages supported by the JVM.
combinations for numbering	Combinations of values of the format token, the language, and the ordinal attribute for numbering	XSLT 2.0 Section 12.3		The processor only supports the ordinal value of yes. Any other values for the ordinal attribute are ignored as if the ordinal attribute was not specified.
data-type for sort	Effect of the data-type attribute for xsl:sort If this has the effective value text, the atomized sort key values are converted to strings before being compared. If it has the effective value number, the atomized sort key values are converted to doubles before being compared. The conversion is done by using the string-FO or number-FO function as appropriate. If the data-type attribute has any other effective value, the value must be a lexical QName with a non-empty prefix and the effect of the attribute is implementation defined.	XSLT 2.0 Section 13.1.2		The supported values for the data-type attribute are 'text' and 'number'. For any other valid values, a warning message is issued and the data-type attribute is ignored.
collation uri	Manner in which the URI is associated with an actual collation rule or algorithm The lang and case-order attributes are ignored if a collation attribute is present. But in the absence of a collation attribute, these attributes provide input to an implementation-defined algorithm to locate a suitable collation. Facilities in XSLT 2.0 and XPath 2.0 that require strings to be ordered rely on the concept of a named collation. A collation is a set of rules that determine whether two strings are equal and, if not, which of them is to be sorted before the other. A collation is identified by a URI, but the manner in which this URI is associated with an actual rule or algorithm is implementation defined	XSLT 2.0 Section 13.1.3		Collation URIs are associated with a Java Collator through the API using one of the `XDynamicContext.bindCollation` methods.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
lang case-order determine collation	Which lang and case-order attributes determine a collation algorithm for sort	XSLT 2.0 Section 13.1.3		The processor retrieves a locale based on the value of the lang attribute. It then creates a Java collator using the `java.text.Collator.getInstance(Locale)` method. The compare method of the collator is overridden to handle the case-order attribute if specified. If the lang attribute was not specified, the processor proceeds as previous using the default locale returned by the Java method `Locale.getDefault()`.
default collation for sort	Default collation for sort If none of the collation, lang or case-order attributes is present in xsl:sort, the collation is chosen in an implementation-defined way. It is not required that the default collation for sorting should be the same as the default collation used when evaluating XPath expressions. (see Section 5.4.1: Initializing the Static Context and Section 3.6.1: The default-collation attribute)	XSLT 2.0 Section 13.1.3		The processor uses the default locale returned by the Java method `Locale.getDefault()` to create a Java collator using the `java.text.Collator.getInstance(Locale)` method.
recognized media types	Set of media types recognized by a processor	XSLT 2.0 Section 16.1		The processor only supports XML files.
picture fallback in date formatting	Fallback representation for picture string in the date formatting functions; the format token n, N, or Nn, that indicates that the value of the component is to be output by name, in lowercase, uppercase, or title-case respectively Components that can be output by name include (but are not limited to) months, days of the week, time zones, and eras. If the processor cannot output these components by name for the chosen calendar and language, it must use an implementation-defined fallback representation.	XSLT 2.0 Section 16.5.1		The processor will output [Calendar: AD] as the fallback.
set supported in date formatting functions	Languages, calendars, and countries that are supported in the date-formatting functions	XSLT 2.0 Section 16.5.2, first paragraph		• Supported calendars: AD, ISO, BE • Supported languages and locales: those returned by `java.util.Locale.getAvailableLocales()`
defaults in date formatting functions	Default language, calendar, and country in the date-formatting functions if arguments are omitted or empty	XSLT 2.0 Section 16.5.2, first paragraph		• Default calendar: AD • Default language and locale: those specified by `java.util.Locale.getDefault()`
default language in date formatting functions	Default language in the date-formatting functions If the language argument is omitted, set to an empty sequence, or set to an invalid value or a value that the implementation does not recognize, the processor uses an implementation-defined language.	XSLT 2.0 Section 16.5.2, fourth paragraph		See defaults in date formatting functions.
names abbrev in language in date formatting functions	Names and abbreviations in the date-formatting functions	XSLT 2.0 Section 16.5.2, seventh paragraph		The International Components for Unicode (ICU) website has a Locale Explorer that contains links to tables for each supported locale. The processor uses the short names for the names of the days of the week and the months specified in those tables as the abbreviated forms, and it uses the long names as the unabbreviated forms.
calendar behavior in date formatting functions	Behavior of calendar in date-formatting functions	XSLT 2.0 Section 16.5.2, tenth paragraph		The processor does not support any calendars with a non-null namespace URI. If the QName supplied has a prefix, the AD calendar will be used.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
default calendar in date formatting functions	Default calendar in date-formatting functions	XSLT 2.0 Section 16.5.2, eleventh paragraph		See defaults in date formatting functions.
supported calendars in date formatting functions	Set of calendars supported in date-formatting functions	XSLT 2.0 Section 16.5.2, first paragraph following table of calendars		See set supported in date formatting functions.
system property values	Actual values returned from system properties	XSLT 2.0 Section 16.5.5, second paragraph following the bulleted list		The value of the system property: • xsl:version is 2.0 • xsl:vendor is IBM Corporation • xsl:vendor-url is http://www.ibm.com/ • xsl:product-name is XL Transform and Query Engine for Java • xsl:product-version is 2.0 • xsl:is-schema-aware is yes • xsl:supports-serialization is yes • xsl:supports-backwards-compatibility is yes • xsl:supports-namespace-axis is yes
set of system properties	Set of system properties	XSLT 2.0 Section 16.6.5 and Erratum XT.E14		The processor supports the following: • All the system properties defined in Section 16.6.5 of the XSLT 2.0 specification • New optional xsl:supports-namespace-axis property defined by XSLT Erratum XT.E14
xslmessage	Way in which an xsl:message instruction sends a message	XSLT 2.0 Section 17, first paragraph		According to Section 17 of the XSLT 2.0 specification, xsl:message sends a message in an implementation-defined way to an implementation-defined destination. The processor uses the string value of the document node created by the xsl:message instruction as the message. If the user did not supply an instance of `com.ibm.xml.xapi.XMessageHandler` on the `com.ibm.xml.xapi.XDynamicContext` for the transformation, the message is written to the System.err output stream; and if the effective value of the terminate attribute was yes, the transformation throws an exception. If the user supplied an instance of `com.ibm.xml.xapi.XMessageHandler`, the message is passed on that object's `report()` method. See the API documentation for more information.
xslmessage destination	Output destination for document created by an xsl:message instruction	XSLT 2.0 Section 17, first paragraph		See xslmessage.
error handling in ext functions	Consequences of an error for an extension function to return a string containing characters that are not XML	XSLT 2.0 Section 18.1.2, paragraph following the third note		Section 18.1.2 of the XSLT 2.0 specification⌂ permits several behaviors when an extension function returns a string containing characters that are not permitted in XML, including treating them as if they were permitted. That is the behavior that the processor implements.
external objects	Way in which external objects are represented in the type system	XSLT 2.0 Section 18.1.3, second paragraph		Data objects whose values are of types that do not map to built-in types as described in “Mapping XML types to Java types” on page 1858 are not supported.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
final result tree delivery	Way in which a final result tree is delivered to an application	XSLT 2.0 Section 19, third paragraph		As described in Section 19.1 of the XSLT 2.0 specification, an implementation may allow a final result tree to be serialized or provide additional mechanisms through which they may be processed. In the processor, a final result tree may be serialized or it may delivered to an application in the form of a `java.xml.transform.Result` object or a `com.ibm.xml.xapi.XSequenceCursor` object. If the URI of the final result tree is not equal to the base output URI, and: • the user supplied a `com.ibm.xml.xapi.XResultResolver` on the `com.ibm.xml.xapi.XDynamicContext`, the result tree will be delivered to the application in the form of the `javax.xml.transform.Result` object returned by `com.ibm.xml.xapi.XResultResolver.getResult()`; • otherwise, the result tree is serialized to the location specified by the URI using `java.net.URL.openConnection()`. A final result tree whose URI is equal to the base output URI is delivered to the application using any `javax.xml.transform.Result` object specified on the `com.ibm.xml.xapi.XExecutable.execute` method used to invoke the transformation, or as the `com.ibm.xml.xapi.XSequenceCursor` returned from the `com.ibm.xml.xapi.XExecutable.execute()` method if that method's signature contains no `javax.xml.transform.Result` argument. If the result tree is delivered in the form of a `javax.xml.transform.stream.StreamResult`, it is serialized to the location specified by the `StreamResult`; if it is delivered as an instance of any other subclass of `Result` or as an `XSequenceCursor`, the result tree is delivered in the form appropriate for that API. Refer to the API documentation for additional information.
URI restriction	Restrictions on form of absolute URI used in an href	XSLT 2.0 Section 19.1, third paragraph following the second note		As described in Section 19.1 of the XSLT 2.0 specification, an implementation may place implementation-defined limitations on an absolute URI that is the value of an href attribute on an xsl:result-document instruction. With the processor, • If the user supplies an `com.ibm.xml.xapi.XResultResolver on the com.ibm.xml.xapi.XDynamicContext` in which the stylesheet is evaluated, there are no limitations on the form of such an absolute URI. • If there is no user-supplied `XResultResolver`, the only restrictions are that if the href attribute is an absolute URI that: – uses the file protocol, the user must have permission to open an output stream to that file – uses some other protocol, the user must have permission to create an output stream to the URI, using the Java equivalent new `java.net.URL(href).openConnection().getOutputStream()`
final result tree location	Location to which final result trees are serialized	XSLT 2.0 Section 2.3 see base output URI		By default, final result tree URIs will be interpreted relative to the base URI of the main output document. If another location is desired, an `XResultResolver` implementation can be registered with the dynamic context to implement user-defined behaviors. Use the `XDynamicContext.setResultResolver()` method to register the `XResultResolver` implementation. The base URI that is used can also be changed by calling the `XDynamicContext.setBaseOutputURI()` method.
default output encoding	Default value of the encoding attribute in xsl:output	XSLT 2.0 Section 20		The default output encoding is UTF-8.
byte-order-mark for UTF-8	Byte-order-mark for UTF-8	XSLT 2.0 Section 20		The default value is no, meaning that no BOM is produced at the start of UTF-8 files.
additional normalization form	Additional normalization form for serialization	XSLT 2.0 and XQuery 1.0 Serialization Section 19.1 Section 20 Section 5.1.8		No implementation-defined normalization forms are provided.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
output version	Permitted and default value of the version attribute in xsl:output	XSLT 2.0 Section 20		The default for XML or XHTML is 1.0; users may explicitly specify 1.0 or 1.1. For HTML, the default is 4.0.
static-typing in stylesheet	Interaction of XSLT 2.0 stylesheets with the static typing feature of XPath 2.0	XSLT 2.0 and XPath 2.0 XSLT 2.0 Section 21 XPath 2.0 Section F.1		This XSLT processor does not currently have a mechanism for requesting that its XPaths perform static typing.
built in types	Type definitions available within a stylesheet	XSLT 2.0 Section 3.13		The processor is schema aware as defined in Section 3.13 of the XSLT 2.0 specification.
namespace for additional error codes	Namespace that is used if additional error QNames are defined by the implementation	XSLT 2.0 Section 2.9		When appropriate, the processor does include the specified error code or code(s) in the text of its messages within square brackets. Additional errors are generally expressed similarly but without a specification prefix on the error identifier. Error conditions are not presented as QNames; applications that wish to respond to errors programmatically must parse these error identifiers off the front of the error-message strings.
unparsed text encoding	Mechanism for determining encoding of an external source	XSLT 2.0 Section 16.2		The processor does not implement any additional heuristics. Encoding resolution proceeds immediately to step 5, and UTF-8 is assumed.
available documents	Mechanism for obtaining a document node and a media type given an absolute URI	XSLT 2.0 Section 2.3 Section 16.1		The default source resolution behavior for documents loaded using the XPath fn:doc function is to resolve relative URIs based on the base URI from the static context. If the base URI is not available, the current working directory is used. Absolute URIs are used unchanged. The default source resolution behavior for documents loaded using the XSLT document function is described in the XSLT 2.0 specification Multiple Source Documents. If no base URI is available, the current working directory is used. If the application needs to either constrain or extend these behaviors, it is possible to plug in an implementation of `XSourceResolver` to the `XDynamicContext` through the `setSourceResolver()` method.
additional output methods	Support for implementation-defined output method	XSLT 2.0 and XQuery 1.0 Serialization Section 3		The processor does not implement any additional output methods.
additional serialization params	Effect of additional serialization parameters on the output of the serializer	XSLT 2.0 and XQuery 1.0 Serialization Section 3 XSLT 2.0 Section 20		The processor implements the following additional serialization parameters: • INDENT AMOUNT: Specifies the number of spaces to use when the indent serialization parameter is enabled. Use the `XOutputParameters.setIndentAmount()` method or the xalan:indent-amount can be specified in the xsl:output element of an XSL stylesheet using the indent-amount attribute. • BC MODE: Enables backwards-compatibility mode for the serializer. If enabled and no output method is specified, XML will be used as a default. If disabled, the rules of XSLT 2.0 will be followed when no output method is specified. Use the `XOutputParameters.setBackwardsCompatible()` method to enable. For XSL stylesheets where the version is less than 2.0, backwards-compatibility mode is enabled by default.
additional normalization-form	Support for additional Unicode Normalization form	XSLT 2.0 and XQuery 1.0 Serialization Section 5.1.8		No implementation-defined normalization forms are provided.
option in encoding phase	Effect of an option that allows the encoding phase to be skipped so that the result of serialization is a stream of Unicode characters	XSLT 2.0 and XQuery 1.0 Serialization Section 4		No such option is provided by the processor.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
cdata sec mechanism	Alternative mechanism requested by the user to create CDATA sections	SR Section 5.1.4 of XSLT 2.0 and XQuery 1.0 Serialization		The processor does not provide an alternative mechanism for creating CDATA sections.
unicode version	Supported version of Unicode	XPath 2.0 Appendix D XQuery 1.0 Appendix D		The processor supports the version of Unicode supported by the host JRE—namely, Unicode Standard Version 4.0 for Java 6 and Unicode Standard Version 6.0 for Java 7.
trace function	Trace function The destination of the trace output is implementation defined.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 4		When using the fn:trace() function, the result will be passed to the `com.ibm.xml.xapi.XMessageHandler` (formerly `com.ibm.xml.xapi.XErrorHandler`). It is up to the user to specify their own `XMessageHandler` to handle the trace results reported by the processor. Trace-message type is indicated by `XMessageHandler.TRACE`.
recovery for under overflow integer operations	Mechanism that allows users to choose between raising an error and returning a result that is modulo the largest representable integer value during overflow or underflow arithmetic operations	XQuery 1.0 and XPath 2.0 Functions and Operators Section 6.2		The processor provides a mechanism to enable overflow detection for integer operations through the `com.ibm.xml.xapi.XStaticContext.setIntegerMathMode(int)` method, which can be called with the constants INTEGER_MATH_MODE_OVERFLOW_DETECTION or INTEGER_MATH_MODE_LIMITED_PRECISION to choose between the two options. A third option is provided to allow arbitrary precision integers.
digits truncation	If the number of digits in the result of a numeric operation exceeds the number of digits that the implementation supports, the result is truncated or rounded in an implementation-defined manner.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 6.2 Section 3.2.3 of XML Schema Part 2: Datatypes Second Edition		The implementation uses the java.math.BigDecimal class, which supports nearly unlimited precision except that the number of digits to the right of the decimal place is limited to Integer.MAX_VALUE. Truncation is only required in the case of division, where there is the possibility of non-terminating decimals. The precision of the fractional part of the result is limited to 18 digits. The rounding mode in this case is ROUND_HALF_UP, where discarded fractions of 0.5 or later are rounded up (away from zero) and lesser fractions are rounded down.
collation abilities	Ability of specified collation to decompose strings into collation units suitable for substring matching is an-implementation-defined-property of a collation.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 7.5		If the Java Collator specified in a call to the `bindCollation` method of the `XDynamicContext` interface is an instance of the `RuleBasedCollator` class, the associated collation URI can be used with one of the functions that performs collation-based substring matching.
year value limits	Maximum number of digits for year values. All minimally conforming processors must support year values with a minimum of 4 digits (YYYY) and a minimum fractional second precision of 1 millisecond or three digits (s.sss); however, conforming processors may set larger implementation-defined-limits on the maximum number of digits that they support in these two situations.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 10.1.1		The year component of the types xs:date, xs:dateTime, xs:gYear, and xs:gYearMonth has the range -(10^9-1) to (10^9)-1.
fractional second precision	Maximum number of digits for fractional second values	XQuery 1.0 and XPath 2.0 Functions and Operators Section 10.1.1		The maximum number of fractional seconds digits supported for the xs:time and xs:dateTime types is 3.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
doc function	Various aspects of the processing provided by fn:doc are implementation defined. Implementations may provide external configuration options that allow any aspect of the processing to be controlled by the user.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 15.5.4		Users may provide an `com.ibm.xml.xapi.XSourceResolver` implementation through the `com.ibm.xml.xapi.XDynamicContext.setSourceResolver(XSourceResolver)` method. The `XSourceResolver` is used by the fn:doc implementation to resolve URIs (thus the user may decide which URIs to support), and its `getSource(String, String)` method must return a JAXP Source object. If no `XSourceResolver` is given, the processor handles the file URI scheme and those supported by the `java.net.URL.openConnection()` method. It does not process URI fragments. There is no built-in support for non-XML media types, but users may use an `XSourceResolver` implementation to provide the processor with an XML representation of non-XML data. DTD validation and schema validation may be applied to the source document depending on the validation setting. The validation setting may be set on `com.ibm.xml.xapi.XFactory` instances using the `setValidating` method, and this setting is inherited by objects created by the `XFactory`. If a `com.ibm.xml.xapi.XDynamicContext` instance is provided to the execute or `executeToList` method of a `com.ibm.xml.xapi.XExecutable` instance, its validation setting is used; otherwise, the setting of the `XExecutable` instance itself is used. The processor does not provide a recovery mechanism for errors in retrieving the resource or parsing or validating its content. The processor does not provide an option to relax the requirement for the fn:doc function to return stable results.
decimal operator precision	For xs:decimal values, the number of digits of precision returned by the numeric operators is implementation defined.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 6.2, last paragraph		The number of digits of precision is the minimum required to represent the exact result without rounding, except in the case of division, where the number of digits of precision for the fractional part of the result is limited to 18.
collection doc stability	User option to evaluate the function without a guarantee of stability in addition to the manner in which such user options are provided	XQuery 1.0 and XPath 2.0 Functions and Operators Sections 15.5.4 and 15.5.6		The processor does not provide an option to relax the requirement for the fn:doc or fn:collection functions to return stable results.
decimal precision	Number of decimal digits supported in xs:decimal The result of casting a string to xs:decimal when the resulting value is not too large or too small but nevertheless has too many decimal digits to be accurately represented, is implementation defined.	XQuery 1.0 and XPath 2.0 Functions and Operators Section 17.1.1		The implementation uses the `java.math.BigDecimal` class, which supports nearly unlimited precision, except that the number of digits to the right of the decimal place is limited to Integer.MAX_VALUE. Memory would be exhausted before this limit is reached.
returns warning	Circumstances in which warnings are returned, and the ways in which warnings are handled	XQuery 1.0 and XPath 2.0 Functions and Operators Section 2.3 XPath 2.0 Section 2.3.1 XQuery 1.0 Section 2.3.1		The user can provide an implementation of `com.ibm.xml.xapi.XMessageHandler` using the `setMessageHandler()(XMessageHandler)` method on either `com.ibm.xml.xapi.XStaticContext` or `com.ibm.xml.xapi.XDynamicContext` for static and dynamic warnings respectively. The `report(int, String, XSourceLocation, Exception)` method of XMessageHandler will be called for each warning, with the first argument equal to `XMessageHandler.WARNING`. If no `XMessageHandler` is provided, warnings are printed to the standard error output stream. Warnings are returned in situations where the processor takes some action to avoid an error condition, such as substituting a default value for an invalid value, that the user might want to correct in the future.

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
report error	Method by which errors are reported to the external processing environment	XQuery 1.0 and XPath 2.0 Functions and Operators Section 2.3 XPath 2.0 Section 2.3.2 XQuery 1.0 Section 2.3.2		The user can provide an implementation of `com.ibm.xml.xapi.XMessageHandler` using the `setMessageHandler()(XMessageHandler)` method on either `com.ibm.xml.xapi.XStaticContext` or `com.ibm.xml.xapi.XDynamicContext` for static and dynamic errors respectively. The `report(int, String, XSourceLocation, Exception)` method of `XMessageHandler` will be called for each error, with the first argument equal to either `XMessageHandler.ERROR` or `XMessageHandler.FATAL_ERROR`. If no `XMessageHandler` is provided, errors are printed to the standard error output stream.
XML 1.0 or XML 1.1 rules	Whether the processor is based on XML 1.0 or XML 1.1 rules for names and supported characters	XPath 2.0 Appendix D XQuery 1.0 Appendix D		The processor follows XML 1.1 rules for the definitions of NCName and supported characters as well as for end-of-line handling and attribute value normalization.
default order of empty sequences	Whether the default handling of empty sequences in "order by" is "empty least" or "empty greatest"	XQuery 1.0 Section 3.8.3		The processor uses the default set by a call to the `XStaticContext` method `setDefaultOrderForEmptySequences`. Otherwise, the default setting is "empty greatest".
pragmas supported	The names and semantics of any XQuery extension expressions (pragmas) supported	XQuery 1.0 Section 3.14	XQST0079	The processor does not recognize or support any extension expressions.
options supported	The names and semantics of any XQuery option declarations supported	XQuery 1.0 Section 4.16		The processor currently supports the java-extension option for declaring extension functions within a query. See the API documentation for more information.
external function parameter passing	Protocols supported for passing parameters to external functions from XQuery and for returning a result	XQuery 1.0 Section 4.15		Extension functions are supported through the processor's `XStaticContext.declareFunction` and `XDynamicContext.bindFunction` methods. See the API documentation for more information.
invoking serialization	The means by which serialization is invoked for the result of evaluating an XQuery query	XQuery 1.0 Section 2.2.4		If an instance of the `javax.xml.transform.stream.StreamResult` interface is supplied on an evaluate method of an `XQueryExecutable` instance, the processor will serialize the result of the query. You can also serialize an `XSequenceCursor` that results from evaluating a query by calling one of the `exportSequence` methods.
default serialization parameters	The default values of the byte-order-mark, encoding, media-type, normalization-form, omit-xml-declaration, standalone, and version serialization parameters	XQuery 1.0 Section 2.2.4 Appendix C.3		The settings of the serialization parameters can be specified on an instance of the XOutputParameters interface. • The default setting of the byte-order-mark serialization parameter is yes for the UTF-16 encoding, and it is no otherwise. • The default setting of the encoding serialization parameter is UTF-8. • The default setting of the media-type serialization parameter is text/xml in the case of the xml output method, text/html in the case of the html and xhtml output methods, and text/plain in the case of the text output method. • The default setting of the normalization-form serialization parameter is none. • The default setting of the omit-xml-declaration serialization parameter is no. • The default setting of the standalone serialization parameter is no. • The default setting of the version serialization parameter is 1.0 for the xml and xhtml output methods and 4.01 for the html output method.
unsuccessful external function call	The effect of an unsuccessful call to an external function	XQuery 1.0 Appendix D		The processor will first report an error to any instance of the `XMessage` handler interface that is supplied and will then throw an exception if a call from XQuery to an external function is unsuccessful for any of (but not limited to) the following reasons: • because the function name is not bound to a function definition using the `XDynamicContext.bindFunction` method • the number or types of arguments supplied does not match the number or types expected by the implementation of the function • some error or exception is thrown by the implementation of the function

Table 252. Implementation-defined behaviors (continued).

This table lists XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-defined behaviors for the API.

Feature	Description	Specification	Error Code	Documented Behavior
locate module imports	The process by which the specific modules to be imported by a module import are identified if the Module Feature is supported (including processing of location hints if any)	XQuery 1.0 Section 4.11		If an XModuleResolver is registered with the XStaticContext, it is used to get Source objects for the modules. If no XModuleResolver is registered or if it returns null, the processor attempts to load one module for each location hint. Relative URIs are resolved against the base URI from the static context if available. If the base URI is not available, relative location hints are interpreted as file paths relative to the current working directory. Absolute location hints are used as is. If a module cannot be loaded for a particular location hint, the processor moves on to the next hint. An error message is emitted only if no modules can be loaded for a namespace.

The following is implementation-specific information related to XSLT 1.0 support:

- The system-property function does not support Java system properties
- If the value of an attribute contains a tab, carriage return or line-feed character with the HTML output method, the processor will serialize the actual character rather than a character reference as part of the attribute value. Either is permitted by the XSLT 2.0 and XQuery 1.0 Serialization Recommendation and by the HTML 4.01 Recommendation, but previous versions of IBM XSLT processors would serialize the character as a character reference in such a context.

Conformance statements:

The processor is an implementation of the XSL Transformations (XSLT) Version 2.0 and the XQuery 1.0 W3C recommendations. See section 21 of the XSLT 2.0 recommendation and section 5 of the XQuery 1.0 recommendation for more information about conformance criteria for processors.

It implements the first editions of the XSLT 2.0, XQuery 1.0, and XPath 2.0 recommendations, with the levels of conformance described in this article, as well as all errata published in the proposed edited recommendations of the second editions of the XSLT 2.0, XQuery 1.0, XPath 2.0 and ancillary recommendations.

- XSL Transformations (XSLT) Version 2.0 (Second Edition)
- XQuery 1.0: An XML Query Language (Second Edition)
- XML Path Language (XPath) 2.0 (Second Edition)
- XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)
- XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)
- XSLT 2.0 and XQuery 1.0 Serialization (Second Edition)

This includes support for the fn:element-with-id function and the XSLT xsl:supports-namespace-axis system property.

XSLT 2.0 conformance

The processor conforms to XSLT 2.0 as a schema-aware XSLT processor. It also supports the following optional features of XSLT 2.0:

- serialization feature
- backwards-compatibility feature

For a complete list of implementation-defined features, read “XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-specific behaviors” on page 1839.

XQuery 1.0 conformance

The processor has minimal conformance to XQuery 1.0. It also supports the following optional features of XQuery 1.0:

- full axis feature
- serialization feature
- schema-import feature
- schema-validation feature
- module feature

For a complete list of implementation-defined features, read “XSLT 2.0, XPath 2.0, and XQuery 1.0 implementation-specific behaviors” on page 1839.

Data Model conformance

The processor supports normative construction of an instance of the XQuery/XPath Data Model from an Infoset or from a PSVI. By default, construction of the instance of the Data Model will be from an Infoset. If the setValidating method of an XFactory instance is called with a value of true, any instance of the Data Model that the processor creates will be constructed from PSVI.

For more information, “Performing basic XQuery operations” on page 1867.

The processor supports both XML 1.0 and XML 1.1.

Extension support

The processor supports the following additional extensions:

- indent-amount extension attribute of xsl:output
- selection of EXSLT extension functions
- redirect extension element

xalan:indent-amount extension attribute of xsl:output

If the value of the indent serialization parameter is yes for an explicit or an implicit xsl:result-document instruction in an XSLT stylesheet, the processor will use the value of any indent-amount extension attribute on the associated xsl:output declaration to determine the amount by which indentation should be increased for every level of element nesting in the serialized result.

The indent-amount extension attribute is in the http://xml.apache.org/xalan namespace.

EXSLT extension functions

In order to facilitate migration of XSLT 1.0 stylesheets, the processor supports many extension functions defined by the EXSLT community initiative. In many cases, these functions duplicate functions that have been included in XSLT 2.0, XPath 2.0 and XQuery 1.0.

For more information about EXSLT, see the EXSLT website.

EXSLT common functions

The processor supports only the node-set common extension function. This function is made redundant by the fact that XSLT 2.0 does not restrict the operations that can be performed on temporary trees.

The EXSLT common functions are in the namespace http://exslt.org/common.

EXSLT dates-and-times functions

The EXSLT dates-and-times functions provide facilities for manipulating date and time values. Most of these functions are redundant with the inclusion of the new date and time data types from XML Schema in XSLT 2.0, XQuery 1.0, and XPath 2.0.

The EXSLT dates-and-times functions are in the namespace http://exslt.org/dates-and-times.

EXSLT dynamic functions

The processor supports only the evaluate dynamic extension function.

The EXSLT dynamic functions are in the namespace http://exslt.org/dynamic.

EXSLT math functions
The EXSLT math functions provide facilities for several commonly used mathematical operations. Only the math:abs, math:max, math:min, and math:highest functions have been made redundant in XSLT 2.0, XQuery 1.0, and XPath 2.0.

The EXSLT math functions are in the namespace http://exslt.org/math.

EXSLT set functions
The EXSLT set functions define facilities for performing set operations on sequences of nodes. These have been made redundant by the new intersect and except set operations and the << and >> node comparison operations introduced in XSLT 2.0, XQuery 1.0, and XPath 2.0.

The EXSLT set functions are in the namespace http://exslt.org/sets.

EXSLT string functions
The EXSLT string functions provide facilities for string manipulation. The tokenize and split functions have been made redundant by the new operations for string manipulation in XSLT 2.0, XQuery 1.0, and XPath 2.0, including the fn:tokenize function and the xsl:analyze-string instruction.

The EXSLT string functions are in the namespace http://exslt.org/strings.

xalan:redirect extension element
The redirect extension element provides a means of directing output from an XSLT stylesheet to more than one output destination. This extension element is made redundant by the new xsl:result-document instruction of XSLT 2.0.

The redirect extension element is in the http://xml.apache.org/xalan namespace.

Choosing between the compiler and the interpreter

You can use either the compiler and the interpreter for preparing and executing an XQuery expression, XPath expression, or XSLT stylesheet. Choosing which one to use is very application specific and depends on several factors.

About this task

This article is about choosing between the compiler and interpreter when you are preparing the expression, query, or stylesheet during the application run time.

Expressions, queries, and stylesheets can also be prepared ahead of time (precompiled). This is the most efficient option because the preparation is done ahead of time instead of during the application run time, but precompiling might not be applicable to all applications. See the articles related to precompiling for more details.

Procedure

Use the setUseCompiler(true) method on the XStaticContext to use the compiler, and use the setUseCompiler(false) method to use the interpreter.
The default is to use the interpreter for preparing an XQuery expression, XPath expression, or XSLT stylesheet.
It takes longer to prepare a compiled executable than an interpreted executable, but a compiled executable generally runs faster; therefore, there is a trade off between the cost of preparing a compiled executable and the improved execution-time efficiency.

Table 253. Differences between using the compiler and the interpreter.

Consider these factors when choosing between compiling and interpreting.

Factor	Description
Number of input documents that the executable will be used to process	If the executable will be used only to process a few input documents, it might not be worth the extra time needed to create a compiled executable because the improved efficiency of the executable might not make up for the extra prepare time. If the executable will be used to process many input documents, it might be worth the extra prepare time.
Size of the input documents	Larger input documents take longer to process; therefore, it might be worth the extra prepare time to create a compiled executable in order to get a more efficient executable object to handle the larger documents.
Size of the expression, query, or stylesheet	It takes longer to prepare a larger expression, query, or stylesheet; therefore, this affects the preparation time and execution time trade off.

Example

```
// Create a new XFactory
XFactory factory = XFactory.newInstance();

// Create a new XStaticContext
XStaticContext staticContext = factory.newStaticContext();

//Use the compiler
sc.setUseCompiler(true);

//Use the interpreter
sc.setUseCompiler(false);
```

Using static and dynamic contexts

You can use the two context interfaces that the XML API provides—XStaticContext and XDynamicContext.

About this task

Static context

The static context is used to configure prepare-time characteristics.

Note: Prepare-time refers to the execution of one of the prepare methods on XFactory or the execution of one of the compile methods on XCompilationFactory.

Static context defines items that are needed to prepare executables, items such as the names and types of external variables and functions that will be available at run time as well as compilation modes like backwards compatibility, math mode, and so on. These items do not change across invocations.

Dynamic context

The dynamic context is used to configure execution-time characteristics.

Dynamic context defines items that are unique to each invocation of an executable, items such as the values for external variables, external function implementations, and resolvers to external inputs or results. These items might change across invocations.

Prepare-time characteristics are not set directly on the XFactory instance so that it can be thread safe. The same is true for execution-time characteristics; they are kept in a separate object from the XExecutable instance so that the executable object itself is thread safe.

The prepare and execute steps themselves are separate because preparation takes time and it would be inefficient to prepare for every execution. Having separate steps allows an expression, query, or stylesheet to be prepared once and the resulting executable object then can be used to process any number of input documents.

Procedure

- Use the static context, XStaticContext, to configure prepare-time characteristics.

Prepare-time characteristics are built directly into the executable object; therefore, after an executable object has been created with a certain set of characteristics, they are fixed and cannot be changed. If an executable object for the same expression, query, or stylesheet is required that has different characteristics, a new one must be generated.

Examples of methods on XStaticContext used for setting prepare-time characteristics:

setUseCompiler
The executable object that is generated is very different for compiled and interpreted; therefore, this is a prepare-time characteristic.

declareVariable
The type of a variable affects how the expression, query, or stylesheet gets compiled and therefore is a prepare-time characteristic.

On the other hand, binding the value of a variable is an execution-time characteristic, and the value can be different for each execution.

setSourceResolver
A source resolver registered at prepare time is used to resolve includes and imports.

A source resolver can also be registered at execution time, but it is used for a different purpose.

- Use the dynamic context, XDynamicContext, to configure execution-time characteristics.

  Execution-time characteristics can be different on each call to the execute method of an executable object.

  Examples of methods on XDynamicContext used for setting execution-time characteristics:

  bind methods
  The bind methods on XDynamicContext interface are used to supply a value for a variable. The value can be different for each execution.

  setSourceResolver
  A source resolver registered at execution time is used to resolve input documents loaded with the XPath fn:doc function or the XSLT fn:document function.

  setXSLTInitialTemplate
  Identify the initial template to invoke for an XSL transformation.

What to do next

The XPath, XQuery, and XSLT specifications also have the concepts of static and dynamic context. For more information, see the following web pages:

- XPath Static Context
- XPath Dynamic Context
- XQuery Static Context
- XQuery Dynamic Context
- XSLT Static Context
- XSLT Dynamic Context

The XStaticContext and XDynamicContext interfaces merge settings from all three languages (XPath, XQuery, and XSLT). To find out which settings apply to which language refer to the “Performing Basic Operations” article for that language in the related tasks listed below.

Mapping XML types to Java types

You can use this mapping between XML types and Java types when using external functions and variables. They are recommended mappings only; other types might work subject to type promotion, casting rules, and the range of values representable by the target type.

Procedure

- Use this table to map between built-in types and Java types when using external functions and variables.

Table 254. Built-in and Java types. This table maps built-in types to Java types.

Built-in Types	Java Types
xs:anyURI	java.lang.String
xs:boolean	boolean, java.lang.Boolean
xs:base64Binary	byte[]
xs:hexBinary	byte[]
xs:date	javax.xml.datatype.XMLGregorianCalendar
xs:dateTime	javax.xml.datatype.XMLGregorianCalendar
xs:time	javax.xml.datatype.XMLGregorianCalendar
xs:duration	javax.xml.datatype.Duration
xs:dayTimeDuration	javax.xml.datatype.Duration
xs:yearMonthDuration	javax.xml.datatype.Duration
xs:gDay	javax.xml.datatype.XMLGregorianCalendar
xs:gMonth	javax.xml.datatype.XMLGregorianCalendar
xs:gMonthDay	javax.xml.datatype.XMLGregorianCalendar
xs:gYear	javax.xml.datatype.XMLGregorianCalendar
xs:gYearMonth	javax.xml.datatype.XMLGregorianCalendar
xs:decimal	java.math.BigDecimal
xs:integer	java.math.BigInteger
xs:nonPositiveInteger	java.math.BigInteger
xs:negativeInteger	java.math.BigInteger
xs:long	long, java.lang.Long
xs:int	int, java.lang.Integer
xs:short	short, java.lang.Short
xs:byte	byte, java.lang.Byte
xs:nonNegativeInteger	java.math.BigInteger
xs:unsignedLong	java.math.BigInteger
xs:unsignedInt	long
xs:unsignedShort	int
xs:unsignedByte	short
xs:positiveInteger	java.math.BigInteger
xs:double	double, java.lang.Double
xs:float	float, java.lang.Float
xs:QName	javax.xml.namespace.QName
xs:NOTATION	javax.xml.namespace.QName
xs:string	java.lang.String
xs:normalizedString	java.lang.String
xs:token	java.lang.String
xs:language	java.lang.String
xs:NMTOKEN	java.lang.String
xs:Name	java.lang.String
xs:NCName	java.lang.String
xs:ID	java.lang.String
xs:IDREF	java.lang.String
xs:ENTITY	java.lang.String
xs:untypedAtomic	java.lang.String
List	com.ibm.xml.xapi.XItemView[]
Union	com.ibm.xml.xapi.XItemView

Table 254. Built-in and Java types (continued). This table maps built-in types to Java types.

Built-in Types	Java Types
All of the above-listed built-in types	com.ibm.xml.xapi.XItemView
	com.ibm.xml.xapi.XSequenceCursor
Complex types (types that represent nodes in the XML document)	com.ibm.xml.xapi.XItemView
	com.ibm.xml.xapi.XSequenceCursor
	org.w3c.dom.Node

The mapping also applies to retrieving values from an XItemView. The XItemView getDoubleValue method returns Java primitive double; the getDateValue, getTimeValue, and getDateTimeValue methods all return an XMLGregorianCalendar; and so on.

If you want to use an external function in your XPath or XQuery expressions and it takes a built-in type as the argument, the actual Java method signature can specify any of the indicated Java types. If you want to use an external function, my:power(arg1 as xs:int, arg2 as xs:int) for example, to calculate the value of the first argument raised to the power of the second argument, you could write a Java method taking two Java primitive int arguments that performs the calculation.

- Use this table to map between sequence types and Java types when using external functions and variables.

Table 255. Sequence and Java types. This table maps sequence types to Java types.

Sequence Types	Java Types
All sequence types	com.ibm.xml.xapi.XSequenceCursor
Sequence type known to be a singleton (a sequence containing only one item)	com.ibm.xml.xapi.XItemView
	One of the types listed in the built-in types to Java types mapping table
Sequence type known to contain only nodes and no atomic items	org.w3c.dom.NodeList
	org.w3c.dom.traversal.NodeIterator

Performing basic XPath operations

You can use the XPathExecutable instances that are created using XFactory.prepareXPath methods to evaluate XPath expressions.

About this task

XPath expressions can be passed to the XFactory.prepareXPath method using a JAXP StreamSource object or using a plain Java string object. The resulting XPathExecutable instance is thread safe and can be reused to evaluate an XPath expression on multiple XML input documents.

Procedure

- Use the XStaticContext object with the prepareXPath method to pass in settings that affect how the XPath expression is prepared.

  If no XStaticContext object is provided, the default setting is used.

  XStaticContext settings relevant to XPath:

  declareFunction
  Declare a Java extension function

  declareNamespace
  Declare a namespace prefix and associate it with a namespace URI to be used as part of an XPath expression

  declareVariable
  Declare a variable externally to be used as part of an XPath expression

  setBaseURI
  Set the base URI used in the resolution of relative URIs (such as by the fn:resolve-uri function)

The default is the base URI of the expression, query, or stylesheet if the base URI is available. If the base URI is not available, the current working directory is used.

setDefaultCollation
Set the default collation URI for string comparison operations

The default is the Unicode code point collation URI.

setDefaultElementTypeNamespace
Specify the URI of the default element or type namespace

Use an empty string to make it unspecified.

The default is an empty string.

setDefaultFunctionNamespace
Specify the URI of the default function namespace

Use an empty string to make it unspecified.

The default is "http://www.w3.org/2005/xpath-functions".

setIntegerMathMode
Set the integer math mode to one of the following:

INTEGER_MATH_MODE_LIMITED_PRECISION
Limit to 18-digit precision for xs:integer (default)

INTEGER_MATH_MODE_ARBITRARY_PRECISION
Allow users to have arbitrary precision for xs:integer

INTEGER_MATH_MODE_OVERFLOW_DETECTION
Generate an error if an overflow is detected

setMessageHandler()
Set the message handler for prepare-time errors

The default behavior is to print errors, warnings, and informational messages to System.err and to generate an XProcessException in the case of an unrecoverable error when compilation cannot continue. If the message handler that is registered does not generate its own exception in the case of an unrecoverable error, an XProcessException occurs.

setUseCompiler
Indicate whether to generate a compiled executable or an interpreted executable

A compiled executable takes longer to generate but runs faster than an interpreted executable. The default is interpreted.

setXPathCompatibilityMode
Sets the compatibility mode to be used when evaluating XPath expressions

The options are as follows:

XPATH2_0_PURE_COMPATIBILITY (default)
Use the default behavior defined by XPath 2.0 in evaluating expressions.

XPATH1_0_BC_COMPATIBILITY
Use the backwards-compatibility behavior defined by XPath 2.0 that yields results that are most consistent with those that would be produced by an XPath 1.0 processor.

XPATH_LATEST_VERSION
Use the default behavior defined by the most recent version of XPath supported by the processor.

- Provide an XDynamicContext object that can be used to pass settings to the XPathExecutable.execute method.

These settings affect the behavior during the execution when the processor is evaluating an XPath expression.

If no XDynamicContext object is provided, the default setting is used.

XDynamicContext settings relevant to XPath:

bind Bind a value to an external variable

bindCollation
Binds an instance of the java.text.Collator class to a particular collation URI for use in string comparisons that use that collation URI

bindFunction
Bind an external function so that it can be used as part of an XPath expression

bindSequence
Bind a sequence value to an external variable

setCollectionResolver
Set the collection resolver used to resolve documents loaded with the XPath fn:collection function

The default behavior is for references to the fn:collection function to return an empty sequence.

setImplicitTimeZone
Set the implicit time zone

The default is to use the system time zone as retrieved through the Java method java.util.TimeZone.getDefault().

setMessageHandler()
Set the message handler for excution-time errors

The default behavior is to print errors, warnings and informational messages to System.err and to generate an XProcessException in the case of an unrecoverable error. If the message handler that is registered does not generate its own exception in the case of an unrecoverable error, an XProcessException occurs.

setSourceResolver
Set the source resolver used to resolve documents loaded with the XPath fn:doc function

The default behavior is to resolve documents based on the base URI from the static context. If the base URI is not available, the current working directory is used.

- Use schemas with XPath expression evaluation.

  For schema-aware XPath evaluation and validating the input XML file, schema files can be registered using the XFactory.registerSchema() method or you can declare the xsi:schemaLocation directive in their input XML files. In either case, validation needs to be enabled using the XFactory.setValidating() method.

Example

The following is a basic example of preparing and executing an interpreted XPath expression.

```
// Create a string for the XPath expression
String expression = "/doc/something";

// Create the factory
XFactory factory = XFactory.newInstance();

// Create an XPath executable for the expression
XPathExecutable xPathExecutable = factory.prepareXPath(expression);

// Create the input XML source
String xml = "<doc><something>something is selected</something></doc>";

// Execute the expression and store the results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xPathExecutable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())));
```

The following is a basic example of preparing and executing a compiled XPath expression.

```
// Create a string for the XPath expression
String expression = "/doc/something";

// Create the factory
XFactory factory = XFactory.newInstance();

// Create a new static context from the factory
XStaticContext xStaticContext = factory.newStaticContext();

// Set the mode to compile for the processor
xStaticContext.setUseCompiler(true);

// Create an XPath executable for the expression
XPathExecutable xPathExecutable = factory.prepareXPath(expression, xStaticContext);

// Create the input XML source
String xml = "<doc><something>something is selected</something></doc>";

// Execute the expression and store the results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xPathExecutable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())));
```

The following is a basic example of preparing and executing interpreted XPath expressions with schema validation.

```
// Create a string for the XPath expression
String expression = "/doc/byte cast as my:derived1-byte-enumeration-Type";

// Create the factory
XFactory factory = XFactory.newInstance();

// Create the schema source
String schema = "<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'" +
          " targetNamespace='http://www.schematype.ibm.com/UDSimple'" +
          " xmlns:my='http://www.schematype.ibm.com/UDSimple'" +
          " xmlns:smokey='http://www.schematype.ibm.com/UDSimple'>"
          + "  <xsd:simpleType name='derived1-byte-enumeration-Type'>"
          + "   <xsd:restriction base='xsd:byte'>"
          + "     <xsd:enumeration value='1' />"
          + "     <xsd:enumeration value='-1' />"
          + "     <xsd:enumeration value='0' />"
          + "     <xsd:enumeration value='127' />"
          + "     <xsd:enumeration value='-128' />"
          + "     <xsd:enumeration value='32' />"
          + "     <xsd:enumeration value='-32' />"
          + "     <xsd:enumeration value='8' />"
          + "     <xsd:enumeration value='-8' />"
          + "     <xsd:enumeration value='2' />"
          + "     <xsd:enumeration value='-2' />"
          + "   </xsd:restriction>"
          + "  </xsd:simpleType>"
          + "</xsd:schema>";
// Load schema
factory.registerSchema(new StreamSource(new ByteArrayInputStream(schema.getBytes())));

// Turn on validation
factory.setValidating(true);

// Create a new static context from the factory
XStaticContext xStaticContext = factory.newStaticContext();

// Add new namespace
xStaticContext.declareNamespace("my", "http://www.schematype.ibm.com/UDSimple");

// Create an XPath executable for the expression
XPathExecutable xPathExecutable = factory.prepareXPath(expression, xStaticContext);

// Create the input XML source
String xml = "<doc>" +
             "    <byte>1</byte>" +
             "</doc>";

// Execute the expression and store the results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xPathExecutable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())));
```

Performing basic XSLT operations

You can use the XSLTExecutable instances that are created using XFactory.prepareXSLT methods to perform XSLT transformations.

About this task

XSLT stylesheets can be passed to the XFactory.prepareXSLT method using a JAXP Source object. The resulting XSLTExecutable instance is thread safe and can be reused to transform multiple input documents.

Procedure

- Use the XStaticContext object with the prepareXSLT method to pass in settings that affect how the XSLT stylesheet is prepared.

  If no XStaticContext object is provided, the default settings are used.

  XStaticContext methods for changing settings relevant to XSLT:

  declareFunction
  Declare a Java extension function

  setBaseURI
  Set the base URI used in the resolution of relative URIs (such as by the fn:resolve-uri function)

  The default is the base URI of the expression, query, or stylesheet if the base URI is available. If the base URI is not available, the current working directory is used.

  setSourceResolver
  Set the source resolver used for includes and imports

  The default source resolution behavior is to use the base URI of the expression, query, or stylesheet if available to resolve imports and includes. If the base URI is not available, the current working directory is used.

  setMessageHandler
  Set the message handler for prepare-time errors and other messages

  The default behavior is to print errors, warnings, and informational messages to System.err and to generate an XProcessException in the case of an unrecoverable error when preparation cannot continue. If the message handler that is registered does not generate its own exception in the case of an unrecoverable error, an XProcessException occurs.

  setIntegerMathMode
  Set the integer math mode to one of the following:

  INTEGER_MATH_MODE_LIMITED_PRECISION (default)
  Use Java long to represent an integer value

  INTEGER_MATH_MODE_ARBITRARY_PRECISION
  Use Java BigInteger to represent an integer value

  INTEGER_MATH_MODE_OVERFLOW_DETECTION
  Use limited precision and generate execution-time errors when overflow is detected

  See the XML API documentation for long and BigInteger for more information about the range of values supported for these types.

  setUseCompiler
  Indicate whether to generate a compiled executable or an interpreted executable

  A compiled executable takes longer to generate but runs faster than an interpreted executable.

  The default is interpreted.

  setDefaultCollation
  Set the default collation used for string comparison operations when the comparison operation does not specify a collation URI explicitly and there is no default collation URI declared within the stylesheet that is in scope for the operation

The default is the Unicode code-point collation URI.

- Provide an XDynamicContext object that can be used to pass settings to the XSLTExecutable.execute method.

  The XDynamicContext object passed in to the XSLTExecutable.execute command defines the settings for that execution.

  If no XDynamicContext object is provided, the default settings are used.

  XDynamicContext methods for changing settings relevant to XSLT:

  bind Bind a value to an XSLT parameter

  If no value is bound, the default value in the stylesheet is used.

  bindSequence
  Bind a sequence of values to an XSLT parameter

  If no value is bound, the default value in the stylesheet is used.

  bindFunction
  Bind a Java method object to an extension function name as declared in the XStaticContext

  setBaseOutputURI
  Set the base output URI to be used when resolving result documents

  The default is to use the base URI of the main result document if available. If the base URI is not available, the current working directory is used.

  setMessageHandler
  Set the message handler for execution-time errors and other messages

  The default behavior is to print errors, warnings and informational messages to System.err and to generate an XProcessException in the case of an unrecoverable error. If the message handler that is registered does not generate its own exception in the case of an unrecoverable error, an XProcessException occurs.

  The message handler registered with the dynamic context is also used for calls to XPath fn:trace and fn:error functions as well as the XSLT xsl:message instruction.

  setImplicitTimeZone
  Set the implicit time zone

  The default is to use the system time zone as retrieved through the Java method java.util.TimeZone.getDefault().

  setResultResolver
  Set the result resolver used to resolve the href specified in xsl:result-document instructions

  The default behavior is to use the base output URI to resolve result documents.

  setSourceResolver
  Set the source resolver used to resolve documents loaded with the XPath fn:doc function and the XSLT fn:document function

  The default behavior is to resolve documents based on the base URI from the static context for fn:doc and for fn:document if no base node is supplied. If the base URI is not available, the current working directory is used.

  setXSLTInitialMode
  Set the initial mode

  If no initial mode is specified, the unnamed default mode is used.

  setXSLTInitialTemplate
  Set the initial template rule

If no initial template rule is specified, the initial template is chosen according to the rules of the xsl:apply-templates instruction for processing the initial context node in the initial mode.

bindCollation

Bind a Java Collator object or a Locale to a collation URI

If the collation URI is referenced in the stylesheet but no Collator or Locale is bound to it (with the exception of the Unicode code-point collation URI, which is bound by default), an error is raised.

setCollectionResolver

Set the collection resolver used to resolve URIs specified in calls to the XPath fn:collection function to a collection of nodes

If the fn:collection function is invoked and no collection resolver is registered an error is raised.

- Achieve an identity transformation by calling the exportItem method on an XItemView object created from a source using the XItemFactory.

  An identity XSLTExecutable cannot be created by preparing a null stylesheet.
- Achieve schema-aware transformations.

  For schema-aware transformations, the processor picks up schemas registered with the XFactory instance through the registerSchema method or through the registerImportedSchemas method of the XSLTExecutable instance. But you should use the xsl:import-schema declaration to import them into the stylesheet as well, because this makes the stylesheet more portable.

  For validating input documents, the schemas can be registered with the XFactory instance or declared in the XML file using the xsi:schemaLocation directive. In either case, validation needs to be enabled using the XFactory class setValidating method

Example

The following is a basic example of preparing and executing an interpreted transformation.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a StreamSource for the stylesheet
StreamSource stylesheet = new StreamSource("simple.xsl");

// Create an XSLT executable for the stylesheet
XSLTExecutable executable = factory.prepareXSLT(stylesheet);

// Create the input source
Source input = new StreamSource("simple.xml");

// Create the result
Result result = new StreamResult(System.out);

// Execute the transformation
executable.execute(input, result);
```

The following is a basic example of preparing and executing a compiled transformation.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a StreamSource for the stylesheet
StreamSource stylesheet = new StreamSource("simple.xsl");

// Create a new static context
XStaticContext staticContext = factory.newStaticContext();

// Enable the compiler
staticContext.setUseCompiler(true);

// Create an XSLT executable for the stylesheet
XSLTExecutable executable = factory.prepareXSLT(stylesheet, staticContext);

// Create the input source
Source input = new StreamSource("simple.xml");

// Create the result
```

```
Result result = new StreamResult(System.out);

// Execute the transformation
executable.execute(input, result);
```

The following is a basic example of creating an identity transformation.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create the item factory
XItemFactory itemFactory = factory.getItemFactory();

// Create the input source
Source input = new StreamSource("simple.xml");

// Create the XItemView object from the input source
XItemView item = itemFactory.item(input);

// Create an XOutputParameters object
XOutputParameters params = factory.newOutputParameters();

// Set parameters
params.setMethod("xml");
params.setEncoding("UTF-8");
params.setIndent(true);

// Create the result
Result result = new StreamResult(System.out);

// Serialize to the result
item.exportItem(result, params);
```

The following is a basic example of creating a schema-aware transformation.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Enable validation
factory.setValidating(true);

// Create the schema source
StreamSource schema = new StreamSource("schema.xsd");

// Register the schema
factory.registerSchema(schema);

// Create the stylesheet source
StreamSource stylesheet = new StreamSource("schema.xsl");

// Create an XSLT executable for the stylesheet
XSLTExecutable executable = factory.prepareXSLT(stylesheet);

// Create the input source
StreamSource input = new StreamSource("schema.xml");

// Create the result
StreamResult result = new StreamResult(System.out);

// Execute the transformation
executable.execute(input, result);
```

Performing basic XQuery operations

You can use the XQueryExecutable instances that are created using XFactory.prepareXQuery methods to evaluate XQuery expressions.

About this task

XQuery expressions can be passed to the XFactory.prepareXQuery method using a JAXP StreamSource object or using a plain Java string object. The resulting XQueryExecutable instance is thread safe and can be reused to evaluate an XQuery expression on multiple XML input documents.

Procedure

- Use the XStaticContext object with the prepareXQuery method to pass in settings that affect how the XQuery expression is prepared.

  If no XStaticContext object is provided, the default setting is used.

  XStaticContext settings relevant to XQuery:

declareFunction
Declare a Java extension function

declareNamespace
Declare a namespace prefix and associate it with a namespace URI to be used as part of an XQuery expression

declareVariable
Declare a variable externally to be used as part of an XQuery expression

setBaseURI
Set the base URI used in the resolution of relative URIs (such as by the fn:resolve-uri function)

The default is the base URI of the expression, query, or stylesheet if the base URI is available. If the base URI is not available, the current working directory is used.

setBoundarySpacePolicy
Specify whether to preserve or strip white space

The options are as follows:

BOUNDARY_SPACE_STRIP (default)

BOUNDARY_SPACE_PRESERVE

setConstructionMode
Specify whether to preserve the type of element and attribute nodes or strip them

The options are as follows:

CONSTRUCTION_MODE_PRESERVE (default)

CONSTRUCTION_MODE_STRIP

setCopyNamespacesModeInherit
Specify whether to inherit or ignore the namespace when copying element nodes

The options are as follows:

COPY_NAMESPACES_MODE_INHERIT (default)

COPY_NAMESPACES_MODE_NO_INHERIT

setCopyNamespacesModePreserve
Specify whether to preserve or strip the namespace when copying element nodes

The options are as follows:

COPY_NAMESPACES_MODE_PRESERVE (default)

COPY_NAMESPACES_MODE_NO_PRESERVE

setDefaultCollation
Set the default collation URI for string comparison operations

The default is the Unicode code point collation URI.

setDefaultElementTypeNamespace
Specify the URI of the default element or type namespace

Use an empty string to make it unspecified.

The default is an empty string.

setDefaultFunctionNamespace
Specify the URI of the default function namespace

Use an empty string to make it unspecified.

The default is "http://www.w3.org/2005/xpath-functions".

setDefaultOrderForEmptySequences
Set the behavior for default order on empty sequences to greatest or least

The options are as follows:

DEFAULT_ORDER_FOR_EMPTY_SEQUENCES_LEAST (default)

DEFAULT_ORDER_FOR_EMPTY_SEQUENCES_GREATEST

setIntegerMathMode
Set the integer math mode to one of the following:

INTEGER_MATH_MODE_LIMITED_PRECISION
Limit to 18-digit precision for xs:integer (default)

INTEGER_MATH_MODE_ARBITRARY_PRECISION
Allow users to have arbitrary precision for xs:integer

INTEGER_MATH_MODE_OVERFLOW_DETECTION
Generate an error if an overflow is detected

setMessageHandler()
Set the message handler for prepare-time errors

The default behavior is to print errors, warnings, and informational messages to System.err and to generate an XProcessException in the case of an unrecoverable error when compilation cannot continue. If the message handler that is registered does not generate its own exception in the case of an unrecoverable error, an XProcessException occurs.

setModuleResolver
Set the module resolver used for module imports

The default module resolution behavior is to attempt to locate one module for each location hint specified in the module import. The default resolution behavior for each location hint is to resolve relative URIs against the base URI from the static context, if the base URI is available, or to interpret them as file paths relative to the current working directory, if the base URI is not available. Absolute URIs are used unchanged. If a module cannot be located for a location hint, the processor ignores it unless no modules can be loaded for the namespace, in which case the processor emits an error message.

setOrderingMode
Specify whether the results returned by certain path expressions, union, intersect, and except expressions as well as FLWOR expressions that have no order by clause are ordered or unordered

The options are as follows:

ORDERING_MODE_ORDERED (default)

ORDERING_MODE_UNORDERED

setUseCompiler
Indicate whether to generate a compiled executable or an interpreted executable

A compiled executable takes longer to generate but runs faster than an interpreted executable.

The default is interpreted.

- Provide an XDynamicContext object that can be used to pass settings to the XQueryExecutable.execute method.

  These settings affect the behavior during the execution when the processor is evaluating an XQuery expression.

  If no XDynamicContext object is provided, the default setting is used.

  XDynamicContext settings relevant to XQuery:

bind Bind a value to an external variable

bindCollation
Binds an instance of the java.text.Collator class to a particular collation URI for use in string comparisons that use that collation URI

bindFunction
Bind an external function so that it can be used as part of an XQuery expression

bindSequence
Bind a sequence value to an external variable

setCollectionResolver
Set the collection resolver used to resolve documents loaded with the XPath fn:collection function

The default behavior is for references to the fn:collection function to return an empty sequence.

setImplicitTimeZone
Set the implicit time zone

The default is to use the system time zone as retrieved through the Java method java.util.TimeZone.getDefault().

setMessageHandler()
Set the message handler for execution-time errors

The default behavior is to print errors, warnings and informational messages to System.err and to generate an XProcessException in the case of an unrecoverable error. If the message handler that is registered does not generate its own exception in the case of an unrecoverable error, an XProcessException occurs.

setSourceResolver
Set the source resolver used to resolve documents loaded with the XPath fn:doc function

The default behavior is to resolve documents based on the base URI from the static context. If the base URI is not available, the current working directory is used.

- Execute XQuery expressions with schema awareness.

  To validate the input XML files with XQuery, schema files can be registered using the XFactory.registerSchema() method or users can declare the xsi:schemaLocation directive in their input XML files. In either case validation needs to be enabled using the XFactory.setValidating() method. XQuery only supports validating the input XML file for now.

Example

The following is a basic example of preparing and executing an interpreted XQuery expression.

```
// Create a string for the XQuery expression
String expression = "/doc/name[@first='David']";

// Create the factory
XFactory factory = XFactory.newInstance();

// Create the XQueryExecutable
XQueryExecutable xQueryExecutable = factory.prepareXQuery(expression);

// Create the input XML source
String xml = "<doc<name first='John'>Wrong</name><name first='David'>Correct</name></doc>";

// Execute the expression and store the results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xQueryExecutable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())));
```

The following is a basic example of preparing and executing a compiled XQuery expression.

```
// Create a string for the XQuery expression
String expression = "/doc/name[@first='David']";

// Create the factory
XFactory factory = XFactory.newInstance();
```

```
// Create a new static context from the factory
XStaticContext xStaticContext = factory.newStaticContext();

// Set the mode to compile for the processor
xStaticContext.setUseCompiler(true);

// Create the XQueryExecutable
XQueryExecutable xQueryExecutable = factory.prepareXQuery(expression, xStaticContext);

// Create the input XML source
String xml = "<doc><name first='John'>Wrong</name><name first='David'>Correct</name></doc>";

// Execute the expression and store the results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xQueryExecutable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())));
```

The following is a basic example of preparing and executing interpreted XQuery expressions with schema awareness.

```
// Create a string for the XQuery expression
String expression = "/my:doc/name[@first='David']/@first";

// Create the factory
XFactory factory = XFactory.newInstance();

// Create the schema source
String schema = "<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'" +
            " targetNamespace='http://www.schematype.ibm.com/UDSimple'" +
            " xmlns:my='http://www.schematype.ibm.com/UDSimple'" +
            " xmlns:smokey='http://www.schematype.ibm.com/UDSimple'>"
                    +" <xsd:element name='doc'>"
                    +   "<xsd:complexType> "
                    +   "<xsd:sequence>"
                    +   "<xsd:element name='name' minOccurs='0' maxOccurs='unbounded'>"
                    +  "<xsd:complexType>"
                    +   "<xsd:attribute name='first' type='xsd:string' use='optional'/>"
                    +  "</xsd:complexType>"
                    +   "</xsd:element>"
                    +  "</xsd:sequence>"
                    +  "</xsd:complexType>"
                    + "</xsd:element>"
                    + "</xsd:schema>";

// Load the schema
factory.registerSchema(new StreamSource(new ByteArrayInputStream(schema.getBytes())));

// Turn on validation
factory.setValidating(true);

// Create a new static context from the factory
XStaticContext xStaticContext = factory.newStaticContext();

// Add a new namespace
xStaticContext.declareNamespace("my", "http://www.schematype.ibm.com/UDSimple");

// Create the XQueryExecutable
XQueryExecutable xQueryExecutable = factory.prepareXQuery(expression, xStaticContext);

// Create the input XML source
String xml = "<my:doc xmlns:my='http://www.schematype.ibm.com/UDSimple'>" +
             "<name first='John'/><name first='David'/></my:doc>";

// Execute the expression and store the results in an XSequenceCursor
XSequenceCursor xSequenceCursor = xQueryExecutable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())));
```

Viewing the results

After your application has prepared or loaded the XExecutable object for an XPath expression (an XPathExecutable object), an XSLT stylesheet (an XSLTExecutable object), or an XQuery expression (an XQueryExecutable object), you apply the XExecutable object to some input and then do something with the result. The source of the input that you provide and what you would like to do with the result determine which of the execute methods you use.

Procedure

- View XPath results.

  In the case of an XPath expression, the execute methods on the XPathExecutable interface return an instance of the XSequenceCursor class that contains the sequence that results from evaluating the expression with the given context item and dynamic context, if any.

  There is also a set of executeToList methods on the XPathExecutable interface. These methods return the sequence as an instance of the java.util.List<XItemView> interface, where each item in the

sequence that results from evaluating the XPath expression is represented in the list as an instance of the XItemView interface. The entries in that list are in the same order as in the sequence that resulted from evaluating the XPath expression.

The following example shows how to get the cost of each item on a purchase order as an XSequenceCursor.

```
XFactory factory = XFactory.newInstance();
XPathExecutable expr =
    factory.prepareXPath(
        "/purchaseOrder/item/(@unit-price * @quantity)");
XSequenceCursor exprResult = expr.execute(new StreamSource("input.xml"));
```

The following example shows how to get the cost of each item on a purchase order in an instance of the java.util.List interface.

```
XFactory factory = XFactory.newInstance();
XPathExecutable expr =
    factory.prepareXPath(
        "/purchaseOrder/item/(@unit-price * @quantity)");
List<XItemView> exprResult =
   expr.executeToList(new StreamSource("input.xml"));
```

- Use the XItemView interface.

  You can access each item in a sequence using the methods on the XItemView interface. You can use the XItemView.isAtomic() method to determine whether the item is an atomic value or a node. If the item is an atomic value, you can use the getValueType() method on the XItemView interface. This method returns an instance of the enumerated type XTypeConstants.Type. If the atomic value is an instance of a built-in atomic type or a user-defined type derived from a built-in atomic type, the result of the getValueType() method is the enumerated value corresponding to that type. For an atomic value that could be of a user-defined derived type, you also might find it handy to use the XItemView.getValueTypeName() method to determine the precise type of the value.

  Given the type of an atomic value, you then can use the appropriate method to get the value.

```
XFactory factory = XFactory.newInstance();
XPathExecutable expr =
    factory.prepareXPath(
        "sum(/purchaseOrder/item/(@unit-price * @quantity))");
XItemView exprResult =
   expr.execute(new StreamSource(purchaseOrder));
double totalCost = 0.0;

// Decide how to get result based on the type of the value
switch (exprResult.getValueType()) {
case DOUBLE: {
    totalCost = exprResult.getDoubleValue();
    break;
}
case FLOAT: {
    totalCost = exprResult.getFloatValue();
    break;
}
case INTEGER: {
    totalCost = exprResult.getLongValue();
    break;
}
case DECIMAL: {
    totalCost = exprResult.getDecimalValue().doubleValue();
    break;
}
default: {
    System.err.println("Unexpected type for result");
}
}
```

The XItemView interface also extends the XNodeView interface, so if the result of calling the XItemView.isAtomic() method is false – that is, the item is a node – you can use the methods inherited from the XNodeView interface to access information about the node. The XNodeView.getNodeQName() method will return the name of the node and the XNodeView.getKind() method will return a value of the enumerated type XNodeView.Kind indicating what sort of node the item is: XNodeView.DOCUMENT, XNodeView.ELEMENT, and so on.

```
XFactory factory = XFactory.newInstance();

// Prepare an expression to get the first node whose string value matches a given query string
XPathExecutable expr =
    factory.prepareXPath(
        "(/descendant-or-self::node()/(self::node()|@*)[. = 'search'])[1]");
XItemView exprResult =
```

```
   expr.execute(new StreamSource(inputFile));

// Print the kind of node found and its name, if appropriate
switch (exprResult.getKind()) {
case ELEMENT: {
    System.out.print("Element " + exprResult.getNodeQName().toString());
    break;
}
case ATTRIBUTE: {
    System.out.print("Attribute " + exprResult.getNodeQName().toString());
    break;
}
case COMMENT: {
    System.out.print("Comment ");
    break;
}
case PROCESSING_INSTRUCTION: {
    System.out.print("PI " + exprResult.getNodeQName().toString());
}
}
```

You can also explore the tree that contains a node using the XTreeCursor interface.

- View XSLT and XQuery results.

  All the execute and executeToList methods available from the XPathExecutable interface are actually inherited from the XExecutable interface, so they are available for use on instances of the XSLTExecutable interface or the XQueryExecutable interface as well. In the case of an instance of the XQueryExecutable interface, the object the method returns contains the sequence that resulted from evaluating the query. In the case of the XSLTExecutable interface, the sequence contains the document node of the primary result of evaluating the stylesheet, if there is any.

  The XSLTExecutable and XQueryExecutable interfaces also define execute methods that accept an instance of the javax.xml.transform.Result interface. The result object that your application supplies will contain the primary result of evaluating the stylesheet, in the case of the XSLTExecutable interface, or the result of evaluating your query, in the case of XQueryExecutable.

  The following example produces a DOM tree as the result of a transformation, and stores the tree in the instance of the DOMResult class that is passed as an argument to the XSLTExecutable.execute method.

```
XFactory factory = XFactory.newInstance();
XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
DOMResult res = new DOMResult();
style.execute(new StreamSource("purchase.xml"), res);
Node node = res.getNode();
```

  The following example produces an instance of the XSequenceCursor interface as the result of transformation.

```
XFactory factory = XFactory.newInstance();
XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
XSequenceCursor xformResult =
        style.execute(new StreamSource("purchase.xml"));
```

- Use the XResultResolver and XSLT.

  An XSLT 2.0 stylesheet can produce more than one result document by using the xsl:result-document instruction. The primary result of the transformation is produced for any result tree that the stylesheet constructs that is not contained in an xsl:result-document instruction or that is contained in an xsl:result-document instruction that has an href attribute whose effective value is a zero-length string. The primary result is returned to your application using the various means described above.

  If your stylesheet evaluates an xsl:result-document instruction that has an href attribute whose effective value is not a zero-length string, by default that result is written to an output stream. The URI of the output stream is determined by resolving the effective value of the href attribute against the setting of the base output URI in the dynamic context.

  You can override this default behavior by supplying an instance of the XResultResolver interface on the XDynamicContext you use for the transformation. The getResult method on the XResultResolver interface is called if any xsl:result-document instructions are invoked; the result of the getResult method is an instance of the Result interface where the result of the xsl:result-document instruction is directed. Using the XResultResolver interface allows your application to decide on a case by case basis where to direct the result of all the xsl:result-document instructions in your stylesheets. If your application did not

supply an instance of the Result interface for the primary result on the execute method of the XSLTExecutable interface, the getResult method of XResultResolver is called for the primary result as well.

If the getResult method of XResultResolver returns a null reference, the default behavior is restored for that particular result—that is, the result is written to an output stream, in the case of a secondary result, or returned from as the result of the execute method as an instance of the XSequenceCursor interface, in the case of the primary result.

The following example uses an instance of the XResultResolver interface to capture all the results produced by a stylesheet as instances of the XSequenceCursorResult class in the variable allResults. The application then could extract the instance of the XSequenceCursor interface that each instance of the XSequenceCursorResult class contains using the XSequenceCursorResult.getSequenceCursor() method.

```
final ArrayList<XSequenceCursorResult> allResults =
            new ArrayList<XSequenceCursorResult>();
XFactory factory = XFactory.newInstance();
XDynamicContext context = factory.newDynamicContext();

// Create XResultResolver that saves XSequenceCursorResult
// instances in allResults
context.setResultResolver(new XResultResolver() {
        public Result getResult(String href, String base) {
            XSequenceCursorResult result =
                    new XSequenceCursorResult();
            allResults.add(result);
            return result;
        }
    });

XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
style.execute(new StreamSource("purchase.xml"), context);

// All results, including the primary, are now available from the allResults variable.
```

Serializing the results

After your application has evaluated an XPath or XQuery expression or performed a transformation with an XSLT stylesheet, you might want to write the output as an actual XML document represented as a file or as a Java string. The process of rendering results as an XML document is known as serialization.

Procedure

- Serialize an XSequenceCursor.

  Your application can call the XSequenceCursor.exportSequence method to serialize a sequence that is represented by an instance of the XSequenceCursor interface. The arguments on this method are an instance of the javax.xml.transform.Result interface and optionally an instance of the XOutputParameters interface.

  If the instance of the Result interface is also an instance of the javax.xml.transform.stream.StreamResult class, the sequence is serialized as described in the XSLT 2.0 and XQuery 1.0 Serialization Recommendation. The StreamResult object can contain an instance of the java.io.Writer class or the java.io.OutputStream class, where the processor will write the serialized sequence.

  You can create an instance of the XOutputParameters interface by calling XFactory.newOutputParameters() and call the methods on that object to override the default serialization parameter settings.

```
XFactory factory = XFactory.newInstance();
XPathExecutable expr = factory.prepareXPath("/purchaseOrder/item[@price > 1000]");
XSequenceCursor exprResult = expr.execute(new StreamSource(inputFile));

System.out.println("Items purchased costing more than $1000");
if (exprResult != null) {
    // Set indenting in order to pretty-print result
    XOutputParameters params = factory.newOutputParameters();
    params.setIndent(true);
    exprResult.exportSequence(new StreamResult(System.out), params);
} else {
    System.out.println("None found");
}
```

  You can also call one of the getOutputParameters() methods on an instance of the XSLTExecutable interface to get the serialization parameters that are associated with a particular output definition in an

XSLT stylesheet. Use the XSLTExecutable.getOutputParameters(javax.xml.namespace.QName) method to get the serialization parameters for a named output definition or the no-argument XSLTExecutable.getOutputParameters() method to get those of the unnamed output definition. You might want do this to perform some post-processing on the result of the transformation using the instance of the XSequenceCursor interface that the transformation produces before serializing the result. If you change the settings of the serialization parameters in the instance of the XOutputParameters interface returned by one of the XSLTExecutable.getOutputParameters() methods, it will not affect the output definition in the stylesheet.

```
XFactory factory = XFactory.newInstance();
XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
XSequenceCursor xformResult = style.execute(new StreamSource("purchase.xml");

XOutputParameters params = style.getOutputParameters(new QName("my-output-definition"));
params.setMethod(XOutputParameters.METHOD_XHTML);
xformResult.exportSequence(new StreamResult("output.html"), params);
```

Note that according to the XSLT 2.0 and XQuery 1.0 Serialization Recommendation, a serialization error results if the sequence that is to be serialized contains attribute nodes or namespace nodes. If the sequence that you need to serialize might contain attribute or namespace nodes, get the values of those nodes as strings or some other appropriate type and serialize those values instead.

- Serialize a single item.

  You can also serialize just the current item in an instance of the XSequenceCursor interface by using one of the exportItem methods. The exportItem methods are inherited from the XItemView interface, so they can be called on an instance of that interface as well.

  As with the exportSequence method described above, the arguments of the exportItem method are an instance of the javax.xml.transform.Result interface and optionally an instance of the XOutputParameters interface. The effect of calling exportItem is identical to the effect of calling exportSequence with a sequence that consists of just the current item.

- Serialize the result of a transformation or query directly.

  Your application can also serialize the result of an XSLT transformation or XQuery expression directly by supplying an instance of the javax.xml.transform.Result interface on the XSLTExecutable.execute method or XQueryExecutable.execute method. The serialization parameter settings are determined by the attributes of any applicable xsl:output declaration or xsl:result-document instruction in the case of an XSLT stylesheet, and are always the default values in the case of the result of an XQuery expression.

```
XFactory factory = XFactory.newInstance();
XSLTExecutable style = factory.prepareXSLT(new StreamSource("style.xsl"));
style.execute(new StreamSource("purchase.xml"),
              new StreamResult("output.xml"));
```

If your application supplies an instance of the XResultResolver interface on a transformation, your application can direct each final result tree to a different destination.

- Use identity transformation.

  You can use the XML API to transform XML data contained in an instance of javax.xml.transform.Source directly to an instance of a javax.xml.transform.Result. This is often referred to as an identity transformation. See “Performing basic XSLT operations” on page 1863 for an example.

Navigating with XSequenceCursor

The XSequenceCursor interface gives you a view of your sequence data.

About this task

A sequence in the XPath and XQuery Data Model contains zero or more atomic values, nodes, or a mixture of both. An instance of the XSequenceCursor interface always contains at least one item. If a sequence is empty, it is always represented by a null reference.

Procedure

- At any given time, an instance of the XSequenceCursor interface is positioned to give you access to one of the items in a sequence. It is positioned initially at the first item in the sequence; and you can move forward or backward in the sequence, one item at a time, using the toNext() or toPrevious() methods, respectively.

  If the toNext() method or toPrevious() method is able to position the XSequenceCursor instance to the next or previous item in the sequence—that is, if there actually is a next or previous item—the method returns true. If the XSequenceCursor instance is already positioned at the last item in the case of the toNext() method or the first item in the case of the toPrevious() method, the method returns false and the instance of the XSequenceCursor interface remains positioned at the same item as before the call.

  The typical way of processing a sequence that is contained in an instance of the XSequenceCursor interface is as shown in the following example:

```
XFactory factory = XFactory.newInstance();
XPathExecutable expr = factory.prepareXPath("1 to 10");
XSequenceCursor exprResult = expr.execute();
long sum = 0;

// If exprResult is null, it means the result sequence is empty
if (exprResult != null) {
    do {
        // Get each value as a primitive Java long value, and accumulate
        sum = sum + exprResult.getLongValue();

        // Advance exprResult to the next item in the sequence
    } while (exprResult.toNext());
}

System.out.println("Sum is " + sum);
```

  The XSequenceCursor interface extends the XItemView interface. You can use the methods inherited from the XItemView interface to access the value and the type of the item in the sequence at which the XSequenceCursor is currently positioned.

- If your application needs to refer to more than one item in the sequence at the same time, you can call the method XSequenceCursor.getSingletonItem() to get an instance of the XItemView interface containing the data associated with the current item in the instance of the XSequenceCursor interface.

  If you change the position of that instance of the XSequenceCursor interface through a call to the toNext() method or the toPrevious() method, the instance of the XItemView interface that was returned by an earlier call to the XSequenceCursor.getSingletonItem() method will still refer to that earlier item.

  Consider the following example, which counts the number of items in a sequence that have the same type and value as the first item.

```
XFactory factory = XFactory.newInstance();

// Make a path expression whose result contains ordered part number as first item
// and all part numbers used by products in the catalog as the subsequent items
XPathExecutable expr =
    factory.prepareXPath(
        "string(/order/item/@part-num),doc('catalog.xml')/catalog/product/part/string(@part-num)");

// Read the invoice file
XSequenceCursor exprResult = expr.execute(new StreamSource(invoiceFile));

int sameAsFirstCount = 0;

// If exprResult is null, it means the result sequence is empty
if (exprResult != null) {
    // Get the first item in the result sequence
    XItemView firstItem = exprResult.getSingletonItem();

    // currentItem always refers to the current item in the result sequence
    XItemView currentItem = exprResult;
    do {
        // Get the type of the first item
        XTypeConstants.Type itemType = firstItem.getValueType();

        // Ensure the type of the first item is the same as the type of
        // the current item, and compare their values as Java objects
        if (itemType == currentItem.getValueType()
                && firstItem.getObjectValue(itemType)
                        .equals(currentItem.getObjectValue(itemType))) {
            sameAsFirstCount++;
        }

        //Advance exprResult (and currentItem) to the next item in the sequence
```

```
    } while (exprResult.toNext());
}

System.out.println("Number of items same as the first == "+(sameAsFirstCount-1));
```

The variable firstItem is created by the XSequenceCursor.getSingletonItem() method, so it always refers to the first item in the sequence. The variable currentItem contains a reference to the XSequenceCursor object, however, so it is always positioned at the current item in the sequence.

- If you ever need to access the items in the sequence non-sequentially, you might find it convenient to use the exportAsList method on the XSequenceCursor interface.

  This method returns an instance of the java.util.List<XItemView> interface that contains the items in your sequence in sequence order.

Navigating with XTreeCursor

You can use the XTreeCursor interface to view your data.

About this task

Suppose the sequence that results from evaluating an XPath or XQuery expression or an XSLT stylesheet contains nodes. You will find it very convenient to access the contents of those nodes by applying further XPath or XQuery expressions to those nodes. However, you might also choose to navigate through the tree structure associated with a node directly through the XML API.

Procedure

- If the result of calling the XItemView.isAtomic() method is false—that is, the context item is a node—your application can call the XItemView.getXTreeCursor() method to gain direct access to the tree that contains the node.

  The XSequenceCursor interface extends XItemView, so calling the XSequenceCursor.getXTreeCursor() method returns an instance of the XTreeCursor interface that you can use to access the node on which the instance of the XSequenceCursor interface is positioned.
- The XTreeCursor interface also extends the XItemView interface; therefore, you can use the methods inherited from the XItemView interface to access information about the node on which your instance of the XTreeCursor interface is currently positioned.

  In particular, you can use the XTreeCursor.getXTreeCursor() method to create another instance of the XTreeCursor interface that you can use to navigate through the tree independently of the instance of the XTreeCursor interface from which it was created.

Example

The following example shows how you can use the methods on the XTreeCursor interface to navigate through the tree containing your XML data.

```
/* Contents of library.xml

<library>
  <book title='Ulysses'><author><first>James</first><last>Joyce</last></author></book>
  <book title='Ada'><author><first>Vladimir</first><last>Nabokov</last></author></book>
</library>
*/

XItemFactory factory = XFactory.newInstance().getItemFactory();
XItemView item = factory.item(new StreamSource("library.xml"));

// 'tree' is initially positioned at the same node as
// 'item' - that is, the document node of the input
XTreeCursor tree = item.getTreeCursor();

// Position tree cursor to "library" element
tree.toFirstChild();

// Position tree cursor to white-space text node
tree.toFirstChild();

// Position to first "book" element
tree.toNextSibling();
```

```
// Position to white-space text node
tree.toNextSibling();

// Position to second "book" element
tree.toNextSibling();

// Create a second instance of XTreeCursor that is initially
// positioned at the same node as 'tree' - that is, the
// second "book" element
XTreeCursor secondBook = tree.getTreeCursor();

// Position 'tree' to "library" element
tree.toParent();

// Position 'secondBook' to "title" attribute
secondBook.toFirstAttribute();
```

The following example navigates through all the nodes in a tree in a depth-first fashion.

```
XItemFactory factory = XFactory.newInstance().getItemFactory();
XItemView item = factory.item(new StreamSource("library.xml"));

XTreeCursor tree = item.getTreeCursor();
boolean hasMoreNodes = true;

do {
    // Process current node

    if (tree.toFirstAttribute()) {
        do {
            // Process attributes
        } while (tree.toNextAttribute());
        tree.toParent();
    }

    if (tree.toFirstNamespace()) {
        do {
            // Process namespaces
        } while (tree.toFirstNamespace());
        tree.toParent();
    }

    boolean foundNext = false;

    // If the current node has a child, visit it next
    // If there's no child, go to the next sibling
    if (tree.toFirstChild() || tree.toNextSibling()) {
        foundNext = true;

    // If there's no child and no sibling, find an
    // ancestor's sibling instead.
    } else {
        do {
            hasMoreNodes = tree.toParent();
            if (hasMoreNodes) {
                foundNext = tree.toNextSibling();
            }
        } while (hasMoreNodes && !foundNext);
    }
} while (hasMoreNodes);
```

Precompiling

You can use this information to help you to precompile an expression, query, or stylesheet.

Procedure

- Precompile using the command-line tool.
- Precompile in Java.
- Load a precompiled executable.

Precompiling using the command-line tools

You can use the CompileXSLT tool to precompile one or more stylesheets, use the CompileXPath tool to precompile one or more XPath expressions, and use the CompileXQuery tool to precompile one or more XQuery expressions.

About this task

Note: See “Precompiling using ANT tasks” on page 1884 for information about using the TaskCompileXPath, TaskCompileXQuery, and TaskCompileXSLT ANT tasks as alternatives to using the CompileXPath, CompileXQuery, and CompileXSLT commands.

Procedure

- **Compile XSLT**

  Location

  The product includes the following script that sets up the environment and invokes the tool.

  Syntax

  Parameters

  -out *output*

  Uses the name *output* as the base name for the generated classes

  By default, the base name is XSLTModule.

  This option is ignored if compiling multiple stylesheets.

  -dir *directory*

  Specifies a destination directory for the generated classes

  The default is the current working directory.

  -pkg *package*

  Specifies a package name prefix for all generated classes

  The default is the Java default package.

  -func name=*funcName* **type=***funcType* **argtype=***argType*

  Adds a function binding to the static context for a single item

  This simply declares the function, and a method object for the function also must be bound to the dynamic context at execution time.

  funcName

  Specifies the name of the function (expressed localPart,namespaceURI)

  funcType

  Specifies the return type of the function (expressed localPart,namespaceURI)

  argType

  Specifies the types of the function arguments (expressed localPart,namespaceURI) and is optional

  This option can be used multiple times.

  If the value of any option contains a blank space, enclose it in quotation marks.

  This option can be used multiple times.

  For example:

```
-func name=getId,http://example.org type=integer,http://www.w3.org/2001/XMLSchema argtype=string,http://www.w3.org/2001/XMLSchema
```

  -baseURI *URI*

  Specifies the base URI of the containing element

  -imm *int*

  Sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values

  Valid values include:

1 Values need only support the minimum precision required for a minimally conforming processor (18 digits).

2 Values should support an arbitrary number of digits of precision; no overflow should occur.

3 Values need only support the minimum precision required for a minimally conforming processor (18 digits); but any overflow condition should be detected and error FOAR0002 should be raised.

-i Forces the compiler to read the stylesheet from standard in

-v Prints the version of the compiler

-h Prints the usage statement

stylesheet
Full path to a file containing an XSL stylesheet to be compiled

The following is a basic example of compiling a stylesheet using the CompileXSLT tool:

- **Compile an XPath expression**

Location
The product includes the following script that sets up the environment and invokes the tool.

Syntax

Parameters

-out *output*
Uses the name *output* as the base name for the generated classes

By default, the base is XPathModule.

This option is ignored if compiling multiple expressions.

-dir *directory*
Specifies a destination directory for the generated executable

The default is the current working directory.

-pkg *package*
Specifies a package name prefix for all generated classes

The default is the Java default package.

-cpm *mode*
Specifies an alternate XPath compatibility mode.

For example, use 1.0 for compatibility with XPath Version 1.0.

-ns *prefix*=*URI*
Specifies a namespace for use during static processing

If the value of any option contains a blank space, enclose it in quotation marks.

This option can be used multiple times. In the case of multiple -ns arguments with the same prefix, the last one prevails.

-schema *URI*
Specifies any schema document that is used to populate the in-scope schema definitions

This option can be used multiple times.

-func name=*funcName* **type=***funcType* **argtype=***argType*
Adds a function binding to the static context for a single item

This simply declares the function, and a method object for the function also must be bound to the dynamic context at execution time.

funcName
Specifies the name of the function (expressed localPart,namespaceURI)

funcType
Specifies the return type of the function (expressed localPart,namespaceURI)

argType
Specifies the types of the function arguments (expressed localPart,namespaceURI) and is optional

This option can be used multiple times.

If the value of any option contains a blank space, enclose it in quotation marks.

This option can be used multiple times.

For example:

```
-func name=getId,http://example.org type=integer,http://www.w3.org/2001/XMLSchema argtype=string,http://www.w3.org/2001/XMLSchema
```

-var name=*varName* **type=***varType*
Adds a variable binding to the static context for a single item

This simply declares the variable, and a value also must be bound to the XDynamicContext at execution time.

varName
Specifies the name of the variable (expressed localPart[, namespaceURI])

If the variable is in no namespace, the namespace URI should be omitted.

Localpart is a required value.

varType
Specifies the type of the variable (expressed localPart,namespaceURI)

If the value of any option contains a blank space, enclose it in quotation marks.

This option can be used multiple times.

For example:

```
-var name=la,"http://www.ibm.com/Los Angeles" type=boolean,http://www.w3.org/2001/XMLSchema
```

-baseURI *URI*
Specifies the base URI of the containing element

-dnet *URI*
Specifies a default namespace URI for element and type names

The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.

-dnf *URI*
Specifies a default namespace URI for function names

The namespace URI, if present, is used for any unprefixed QName appearing in a position where a function name is expected.

-imm *int*
Sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values

Valid values include:

1 Values need only support the minimum precision required for a minimally conforming processor (18 digits).

2 Values should support an arbitrary number of digits of precision; no overflow should occur.

3 Values need only support the minimum precision required for a minimally conforming processor (18 digits); but any overflow condition should be detected and error FOAR0002 should be raised.

-v Prints the version of the compiler

-h Prints the usage statement

xpathfile
Full path to a file containing an XPath expression to be compiled

-i Forces the compiler to read the XPath expression from standard in

The following is a basic example of compiling an XPath expression using the CompileXPath tool:

- **Compile an XQuery expression**

Location
The product includes the following script that sets up the environment and invokes the tool.

Syntax

Parameters

-out *output*
Uses the name *output* as the base name for the generated classes

By default, the base name is XQueryModule.

This option is ignored if compiling multiple expressions.

-dir *directory*
Specifies a destination directory for the generated classes

The default is the current working directory.

-pkg *package*
Specifies a package name prefix for all generated classes

The default is the Java default package.

-func name=*funcName* **type=***funcType* **argtype=***argType*
Adds a function binding to the static context for a single item

This simply declares the function, and a method object for the function also must be bound to the dynamic context at execution time.

funcName
Specifies the name of the function (expressed localPart,namespaceURI)

funcType
Specifies the return type of the function (expressed localPart,namespaceURI)

argType
Specifies the types of the function arguments (expressed localPart,namespaceURI) and is optional

This option can be used multiple times.

If the value of any option contains a blank space, enclose it in quotation marks.

This option can be used multiple times.

For example:

```
-func name=getId,http://example.org type=integer,http://www.w3.org/2001/XMLSchema argtype=string,http://www.w3.org/2001/XMLSchema
```

-baseURI *URI*
Specifies the base URI of the containing element

-dnet *URI*
Specifies a default namespace URI for element and type names

The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.

-dnf *URI*
Specifies a default namespace URI for function names

The namespace URI, if present, is used for any unprefixed QName appearing in a position where a function name is expected.

-imm *int*
Sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values

Valid values include:

1 Values need only support the minimum precision required for a minimally conforming processor (18 digits).

2 Values should support an arbitrary number of digits of precision; no overflow should occur.

3 Values need only support the minimum precision required for a minimally conforming processor (18 digits); but any overflow condition should be detected and error FOAR0002 should be raised.

-bsp *int*
Specifies the boundary space policy

Valid values include:

1 Preserve white spaces

2 Strip white spaces

This is the default value.

-csm *int*
Specifies the construction mode

Valid values include:

1 Preserve

The type of a constructed element node is xs:anyType, and all attribute and element nodes copied during node construction retain their original types.

This is the default value.

2 Strip

The type of a constructed element node is xs:untyped, all element nodes copied during node construction receive the type xs:untyped, and all attribute nodes copied during node construction receive the type xs:untypedAtomic.

-cnmi *int*
Specifies the inherit part of the copy-namespaces

Valid values include:

1 Inherit

Inherit mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.

This is the default value.

2 No inherit

No-inherit mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.

-cnmp *int*
Specifies the preserve part of the copy-namespaces

Valid values include:

1 Preserve

Preserve mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.

This is the default value.

2 No preserve

No-preserve mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.

-eso *int*
Specifies the empty sequence order

Valid values include:

1 Greatest

2 Least

This is the default value.

-ordm *int*
Specifies the ordering mode

Valid values include:

1 Ordered results are to be returned by certain path expressions, union, intersect, and except expressions, as well as FLWOR expressions that have no order by clause

This is the default value.

2 Unordered results are to be returned by certain path expressions, union, intersect, and except expressions, as well as FLWOR expressions that have no order by clause

-v Prints the version of the compiler

-h Prints the usage statement

xqueryfile
Full path to a file containing an XQuery expression to be compiled

-i Forces the compiler to read the stylesheet from standard in

The following is a basic example of compiling an XQuery expression using the CompileXQuery tool:

Precompiling using ANT tasks

You can use the TaskCompileXPath, TaskCompileXQuery, and TaskCompileXSLT ANT tasks as alternatives to using the CompileXPath, CompileXQuery, and CompileXSLT commands.

About this task

ANT task elements:

argType

This element indicates the types of the function arguments expressed in terms of the localpart and namespaceURI.

Parameters specified as nested elements:

localpart

A nested localpart element must be specified to provide the local part of the qualified name.

namespaceURI

A nested namespaceURI element can be specified to indicate the namespaceURI.

Example:

```
<argType>
  <localpart>boolean</localpart>
  <namespaceURI>"http://www.w3.org/2001/XMLSchema"</namespaceURI>
</argType>
```

baseURI

This element specifies the base URI of the containing element.

Example:

```
<baseURI>"http://www.ibm.com/Los Angeles"</baseURI>
```

bsp This element specifies the Boundary Space Policy.

Valid values include:

- 1 = preserve white spaces
- 2 = strip white spaces

The default value is 2 (strip white spaces).

Example:

```
<bsp>1</bsp>
```

cnmi This element specifies the inherit part of the copy-namespaces.

Valid values include:

- 1 = inherit

  Inherit mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.
- 2 = no inherit

  No-inherit mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.

The default is 1 (inherit).

Example:

```
<cnmi>2</cnmi>
```

cnmp This element specifies the preserve part of the copy-namespaces.

Valid values include:

- 1 = preserve

  Preserve mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.
- 2 = no preserve

No-preserve mode should be used in namespace binding assignment when an existing element node is copied by an element constructor.

The default is 1 (preserve).

Example:

```
<cnmp>2</cnmp>
```

cpm This element specifies an alternate XPath Compatibility Mode.

Valid values are:

- Latest
- 1.0
- 2.0

For example: use 1.0 for compatibility with XPath Version 1.0.

The default is 2.0.

Example:

```
<cpm>1.0</cpm>
```

csm This element specifies the Construction Mode.

Valid values include:

- 1 = preserve

  The type of a constructed element node is xs:anyType, and all attribute and element nodes copied during node construction retain their original types.
- 2 = strip

  The type of a constructed element node is xs:untyped, all element nodes copied during node construction receive the type xs:untyped, and all attribute nodes copied during node construction receive the type xs:untypedAtomic.

The default value is 1 (preserve).

Example:

```
<csm>2</csm>
```

dir This element specifies a destination directory for the executables.

The default is the current working directory.

If the value contains a blank space, enclose it in quotation marks.

Example:

```
<dir>C:/precompiledXSLT</dir>
```

dnet This element specifies a default namespace URI for element and type names.

The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.

Example:

```
<dnet>http://example.org/ibm</dnet>
```

dnf This element specifies a default namespace URI for function names.

The namespace URI, if present, is used for any unprefixed QName appearing in a position where a function name is expected.

Example:

```
<dnf>http://my.org</dnf>
```

eso This element specifies the Empty Sequence Order.

Valid values include:

- 1 = greatest
- 2 = least

The default value is 2 (empty sequences least).

Example:

```
<eso>1</eso>
```

function

For the CompileXSLT, CompileXPath, and CompileXQuery tasks, this element defines and binds a function to the static context for a single item. Note that this simply declares the function, and a Method object for the function must also be bound to the dynamic context. Note: this element can be specified multiple times.

Parameters specified as nested elements:

name A nested name element must be specified to indicate the name of the function.

type A nested type element must be specified to indicate the type of the function.

argType

A nested argType element can be specified to indicate the types of the function arguments.

This element is optional and can be specified multiple times.

Example:

```
<function>
  <name>
    <localpart>la</localpart>
    <namespaceURI>"http://www.ibm.com/Los Angeles"</namespaceURI>
  </name>
  <type>
    <localpart>boolean</localpart>
    <namespaceURI>http://www.w3.org/2001/XMLSchema</namespaceURI>
  </type>
</function>
```

imm This element sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values.

Valid values include:

- 1 = values need only support the minimum precision required for a minimally conforming processor (18 digits)
- 2 = values should support an arbitrary number of digits of precision; no overflow should occur
- 3 = values need only support the minimum precision required for a minimally conforming processor (18 digits) but any overflow condition should be detected and error FOAR0002 raised

Example:

```
<imm>2</imm>
```

inputfile

This element specifies the full path to a file containing an XSL, XPath, or XQuery to be compiled.

Example:

```
<inputfile>C:/XSLT/simple.xsl</inputfile>
```

localpart

This element indicates the local part of a qualified name.

Example:

```
<localpart>la</localpart>
```

name This element indicates the name of the function or variable (expressed in terms of the localpart and namespaceURI).

In the case of a variable that is in no namespace the namespace URI should be omitted.

Parameters specified as nested elements:

localpart
A nested localpart element must be specified to provide the local part of the qualified name.

namespaceURI
A nested namespaceURI element can be specified to indicate the namespaceURI.

Example:

```
<name>
  <localpart>la</localpart>
  <namespaceURI>"http://www.ibm.com/Los Angeles"</namespaceURI>
</name>
```

namespaceURI
This element indicates the namespaceURI part of a qualified name.

If the value contains a blank space. enclose it in quotation marks.

Example:

```
<namespaceURI>"http://www.ibm.com/Los Angeles"</namespaceURI>
```

ns This element specifies a namespace for use during static processing.

The value should be specified as prefix=URI.

If the value contains a blank space, enclose it in quotation marks.

This element can be used multiple times.

Example:

```
<ns>my=http://www.example.com/examples</ns>
```

ordm This element specifies the Ordering Mode.

Valid values include:

- 1 = ordered results are to be returned by certain path expressions, union, intersect, and except expressions, and FLWOR expressions that have no order by clause
- 2 = unordered results are to be returned by certain path expressions, union, intersect, and except expressions, and FLWOR expressions that have no order by clause

The default value is 1 (ordered).

Example:

```
<ordm>2</ordm>
```

out This element specifies the name of the generated executable.

The default executable name is XSLTModule, XPathModule, or XQueryModule depending on the corresponding task.

This option is ignored if compiling multiple files.

Example:

```
<out>sample</out>
```

dir This element specifies a destination directory for the executables. Default is the current working directory.

If the value contains a blank space, enclose it in quotation marks.

Example:

```
<pkg>com.mycompany.precompiled</pkg>
```

schema
This element specifies a schema document that will be used to populate the in-scope schema definitions.

This element can be used multiple times.

Example:

```
<schema>C:/samples/xpath/variousTypesNodeTest.xsd</schema>
```

type This element indicates the type of the function or variable expressed in terms of the localpart and namespaceURI.

Parameters specified as nested elements:

localpart
A nested localpart element must be specified to provide the local part of the qualified name.

namespaceURI
A nested namespaceURI element can be specified to indicate the namespaceURI.

Example:

```
<type>
  <localpart>boolean</localpart>
  <namespaceURI>"http://www.w3.org/2001/XMLSchema"</namespaceURI>
</type>
```

variable

This element defines and binds a variable to the static context for a single item.

This simply declares the variable, and a value must be bound to the dynamic context.

This element can be specified multiple times.

Parameters specified as nested elements:

name A nested name element must be specified to indicate the name of the variable.

type A nested type element must be specified to indicate the type of the variable.

Example:

```
<variable>
  <name>
    <localpart>booleanVar</localpart>
  </name>
  <type>
    <localpart>boolean</localpart>
    <namespaceURI>http://www.w3.org/2001/XMLSchema</namespaceURI>
  </type>
</variable>
```

Procedure

- **Use TaskCompileXPath**

  This task can be used to precompile one or more XPath expressions. The output will be a set of Java classes that subsequently can be used to execute the expressions without the performance overhead of dynamic compilation.

  Parameters specified as nested elements:

  out A nested **out** specifies the name of the generated executable.

  The default executable name is XPathModule.

  dir A nested **dir** specifies a destination directory for the executables.

  The default is the current working directory.

  pkg A nested **pkg** specifies a package name prefix for all generated classes.

  The default is the Java default package.

  cpm A nested **cpm** specifies an alternate XPath Compatibility Mode.

  ns A nested **ns** specifies a namespace for use during static processing.

schema
A nested **schema** specifies a schema document that will be used to populate the in-scope schema definitions.

function
A nested **function** definition can be specified.

This will create a function binding to the static context for a single item.

variable
A nested **variable** definition can be specified.

This will add a variable binding to the static context for a single item.

baseURI
A nested **baseURI** specifies the base URI of the containing element.

dnet A nested **dnet** specifies a default namespace URI for element and type names.

dnf A nested **dnf** specifies a default namespace URI for function names.

imm A nested **imm** sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values.

inputfile
A nested **inputfile** specifies the full path to a file containing XPath expression to be compiled.

Note: This element is required and can be used multiple times.

Example:

```
<target name="testXPath">
  <taskdef name="compileXPath" classname="com.ibm.xml.xapi.ant.TaskCompileXPath"/>
  <compileXPath>
    <out>sample</out>
    <dir>"C:/precompiledXPath"</dir>
    <pkg>com.mycompany.precompiled</pkg>
    <variable>
      <name>
        <localpart>booleanVar</localpart>
      </name>
      <type>
        <localpart>boolean</localpart>
        <namespaceURI>http://www.w3.org/2001/XMLSchema</namespaceURI>
      </type>
    </variable>
    <inputfile>C:/XPath/xpath.txt</inputfile>
  </compileXPath>
</target>
```

- **Use TaskCompileXQuery**

  This task can be used to precompile one or more XQuery expressions. The output will be a set of Java classes that subsequently can be used to execute the expressions without the performance overhead of dynamic compilation.

  Parameters specified as nested elements:

  out A nested **out** specifies the name of the generated executable.

  The default executable name is XQueryModule.

  dir A nested **dir** specifies a destination directory for the executables.

  The default is the current working directory.

  pkg A nested **pkg** specifies a package name prefix for all generated classes.

  The default is the Java default package.

  function
  A nested **function** definition can be specified.

This will create a function binding to the static context for a single item.

baseURI
A nested **baseURI** specifies the base URI of the containing element.

dnet A nested **dnet** specifies a default namespace URI for element and type names.

dnf A nested **dnf** specifies a default namespace URI for function names.

imm A nested **imm** sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values.

bsp A nested **bsp** specifies the Boundary Space Policy.

csm A nested **csm** specifies the Construction Mode.

cnmi A nested **cnmi** specifies the inherit part of the copy-namespaces.

cnmp A nested **cnmp** specifies the preserve part of the copy-namespaces.

eso A nested **eso** specifies the Empty Sequence Order.

ordm A nested **ordm** specifies the Ordering Mode.

inputfile
A nested **inputfile** specifies the full path to a file containing an XQuery expression to be compiled. Note: this element is required and can be used multiple times.

Example:

```
<target name="testXQuery">
  <taskdef name="compileXQuery" classname="com.ibm.xml.xapi.ant.TaskCompileXQuery"/>
  <compileXQuery>
    <out>sample</out>
    <dir>"C:/precompiledXQuery"</dir>
    <pkg>com.mycompany.precompiled</pkg>
    <inputfile>C:/XQuery/xquery.sq</inputfile>
  </compileXQuery>
</target>
```

- **Use TaskCompileXSLT**

  This task can be used to precompile one or more stylesheets. The output will be a set of Java classes that subsequently can be used to execute transformations without the performance overhead of dynamic compilation.

  Parameters specified as nested elements:

  out A nested **out** specifies the name of the generated executable.

  The default executable name is XSLTModule.

  This element is ignored if compiling multiple stylesheets.

  dir A nested **dir** specifies a destination directory for the executables.

  The default is the current working directory.

  pkg A nested **pkg** specifies a package name prefix for all generated classes.

  The default is the Java default package.

  function
  A nested **function** definition can be specified.

  This will create a function binding to the static context for a single item.

  baseURI
  A nested **baseURI** specifies the base URI of the containing element.

imm A nested **imm** sets the integer math mode, which is a constant representing the level of precision required and whether overflow detection is required when working with xs:integer values.

inputfile
A nested **inputfile** specifies the full path to a file containing an XSL stylesheet to be compiled.

Note: This element is required and can be used multiple times.

Example:

```
<target name="testXSLT">
  <taskdef name="compileXSLT" classname="com.ibm.xml.xapi.ant.TaskCompileXSLT"/>
  <compileXSLT>
    <out>sample</out>
    <dir>"C:/precompiledXSLT"</dir>
    <pkg>com.mycompany.precompiled</pkg>
    <function>
      <name>
        <localpart>la</localpart>
        <namespaceURI>"http://www.ibm.com/Los Angeles"</namespaceURI>
      </name>
      <type>
        <localpart>boolean</localpart>
        <namespaceURI>http://www.w3.org/2001/XMLSchema</namespaceURI>
      </type>
    </function>
    <inputfile>C:/XSLT/simple.xsl</inputfile>
  </compileXSLT>
</target>
```

Precompiling in Java

You can use the XCompilationFactory interface and its various compile and load methods to compile an expression, query, or stylesheet in advance. The Java classes can be loaded at execution time, therefore avoiding the cost of compilation in the application run time.

Procedure

- Retrieve the XCompilationFactory by calling the getCompilationFactory method on the XFactory class.

  An XCompilationFactory instance is associated with a particular XFactory instance, so they share registered schemas. If a new schema is registered with the XFactory instance; therefore, it is visible to the associated XCompilationFactory instance.
- Create a new XCompilationParameters instance by calling the XCompilationFactory newCompilationParameters method, passing in the base class name to use for the generated classes.

  Configure the parameters further by using the set methods described in this table.

Table 256. Valid set methods.

These set methods are defined in the XCompilationParameters interface and are valid for generating a precompiled executable.

Set Methods Valid for Generating a Precompiled Executable	Description	Default
setPackageName	Specify the package name for the generated classes. The value must be a valid Java package name.	Java default package
setDirectoryName	Specify the directory to which the generated classes should be written. The directory must exist.	Current working directory as retrieved by calling the Java System.getProperty method with the property user.dir

  The setClassLoader method is only valid when loading the generated classes using one of the load methods. If the class loader is set at compile time, it is ignored.
- Use one of the compile methods on the XCompilationFactory, passing in the XCompilationParameters, to generate the precompiled executable.

  The compile methods use the XStaticContext just like the XFactory prepare methods to configure prepare-time settings. Note that compilation is implied when the compile methods are used; therefore,

changing the use-compiler setting through the XStaticContext setUseCompiler method has no effect on these methods. If no static context is specified, the default settings are used.

Example

The following is a basic example of precompiling an XPath expression.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Get the compilation factory
XCompilationFactory compileFactory = factory.getCompilationFactory();

// Create the compilation parameters
XCompilationParameters params = compileFactory.newCompilationParameters("MyXPath");
params.setPackageName("org.example.myxpath");

// Generate the compiled classes
compileFactory.compileXPath("/doc/item[@id > 3000]", params);
```

Appropriate compile methods are available for XQuery and XSLT as well.

Loading a precompiled executable

You can use the XCompilationFactory interface and its various load methods to load a precompiled expression, query, or stylesheet. These load methods load the Java classes and return an XPathExecutable, XQueryExecutable, or XSLTExecutable object respectively.

Procedure

- Retrieve the XCompilationFactory by calling the getCompilationFactory method on the XFactory class.

  An XCompilationFactory instance is associated with a particular XFactory instance, so they share registered schemas. If a new schema is registered with the XFactory instance, therefore, it is visible to the associated XCompilationFactory instance.
- Create a new XCompilationParameters instance by calling the XCompilationFactory newCompilationParameters method, passing in the base class name of the classes to be loaded.

  Configure the parameters further by using the set methods described in this table.

Table 257. Valid set methods.

These are set methods that are defined in the XCompilationParameters interface and are valid for loading a precompiled executable.

Set Methods Valid for Loading a Precompiled Executable	Description	Default
setPackageName	Specify the package name of the classes to load.	Java default package
setClassLoader	Specify the class loader to use.	Class loader that was used to load the processor

  The setDirectoryName method is not valid when loading a precompiled executable because the classpath is used to search for the classes. If the directory name is set, it is ignored. The setDirectoryName method can be used when generating precompiled executables to specify the directory to which you want to write the classes.
- Use one of the load methods on the XCompilationFactory, passing in the XCompilationParameters, to load the precompiled executable.

Example

The following is a basic example of loading a precompiled XPath expression.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Get the compilation factory
XCompilationFactory compileFactory = factory.getCompilationFactory();

// Create the compilation parameters
XCompilationParameters params = compileFactory.newCompilationParameters("MyXPath");
params.setPackageName( "org.example.myxpath");
```

```
// Load the executable
XPathExecutable executable = compileFactory.loadXPath(params);

// Create the input source
StreamSource input = new StreamSource("simple.xml");

// Execute the XPath expression
XSequenceCursor cursor = executable.execute(input);
```

Appropriate load methods are available for XQuery and XSLT as well.

Using resolvers

You can use this information to help you to use resolvers.

Procedure

- Use source and result resolvers.
- Register a collection resolver.
- Register a schema resolver.
- Use an unparsed text resolver.
- Use a module resolver.

Using source and result resolvers

You can use this information to help you to use source and result resolvers.

Procedure

- Use a source resolver at prepare time.
- Use a source resolver at execution time.
- Use a result resolver at execution time.

Using a source resolver at prepare time:

By specifying a source resolver at the time that an executable is being prepared, you can tell the processor how to interpret the URIs referenced at that time.

Procedure

Specify a source resolver at the time an executable is being prepared.
This tells the processor how to interpret the URIs referenced at that time, in a stylesheet's xsl:import and xsl:include directives for example.
The default source-resolution behavior is to interpret relative URIs in terms of the base URI of the expression, query, or stylesheet if the base URI is available or to interpret them as file paths relative to the current working directory if the base URI is not available. Absolute URIs are used unchanged.
To change this behavior, write a Java class that implements the XSourceResolver interface and register it with the XStaticContext before preparing the stylesheet.
The following is a basic example of how to register the source resolver.

```
XFactory factory = XFactory.newInstance();

// Register the source resolver with the static context
XStaticContext staticContext = factory.newStaticContext();
XSourceResolver sourceResolver=new ASourceResolver(replacementBase);
staticContext.setSourceResolver(sourceResolver);

// Prepare the sylesheet
XSLTExecutable executable = factory.prepareXSLT(new StreamSource(stylesheetFile), staticContext);

XDynamicContext dynamicContext = factory.newDynamicContext();

// Execute the stylesheet
XSequenceCursor cursor = executable.execute(new StreamSource(inputFile), dynamicContext);
```

Using a source resolver at execution time:

By specifying a source resolver at the time that an executable is being executed, you can tell the processor how to interpret the URIs referenced at that time.

Procedure

Specify a source resolver at the time that an executable is being executed.
This tells the processor how to interpret the URIs referenced at that time, in calls to the fn:doc() or document() functions for example.
The default source-resolution behavior is to interpret relative URIs in terms of the base URI of the expression, query, or stylesheet if the base URI is available or to interpret them as file paths relative to the current working directory if the base URI is not available. Absolute URIs are used unchanged.
To change this behavior, write a Java class that implements the XSourceResolver interface and register it with the XDynamicContext before executing the expression, query, or stylesheet.
The following is a basic example of how to register the source resolver.

```
XFactory factory = XFactory.newInstance();

XStaticContext staticContext = factory.newStaticContext();

// Prepare the stylesheet
XSLTExecutable executable = factory.prepareXSLT(new StreamSource(stylesheetFile), staticContext);

XDynamicContext dynamicContext = factory.newDynamicContext();
// Register the source resolver with the dynamic context
XSourceResolver sourceResolver=new ASourceResolver(replacementBase);
dynamicContext.setSourceResolver(sourceResolver);

// Execute the XPath expression
XSequenceCursor cursor = executable.execute(new StreamSource(inputFile), dynamicContext);
```

Using a result resolver at execution time:

By specifying a result resolver at execution time, you can tell the processor how to redirect output URIs specified in the executable.

Procedure

To activate a result resolver, register it with the dynamic context before calling execute().
Result resolvers perform essentially the same function as source resolvers, but on the output side of the processor. They allow you to intercept and redirect output URIs specified in the executable, such as xsl:result-document directives in a stylesheet.
The default resolution behavior is to use the base output URI to resolve result documents if the URI reference is relative. Absolute URIs are used unchanged.
The following is a basic example of a result resolver.

```
class AResultResolver implements XResultResolver
{
    String _replacementBase;

    public AResultResolver(String replacementBase)
    {
      _replacementBase=replacementBase;
    }

    // Resolve URIs by loading the resource as an XSLT stylesheet
    // and evaluating it - return the result as the Source to use
    public Result getResult(String href, String base) {
      String rebasePrefix="rebase://";

      if(href.startsWith(rebasePrefix))
      {
       href=href.substring(rebasePrefix.length());
       base=_replacementBase;
      }

      java.net.URI baseURI;
      Result result=null;
      try {
         // Get base URI object
          baseURI = new java.net.URI(base);
          // Resolved relative reference against base URI
          URI resolvedURI = baseURI.resolve(href);
          // Try to read...
          result = new StreamResult(resolvedURI.toString());
```

```
      } catch (java.net.URISyntaxException use) {
          throw new RuntimeException(use);
     }

      return result;
    }
}
```

The following is a basic example of registering and using the resolver.

```
XFactory factory = XFactory.newInstance();

XStaticContext staticContext = factory.newStaticContext();

// Prepare the stylesheet
XSLTExecutable executable = factory.prepareXSLT(new StreamSource(stylesheetFile), staticContext);

XDynamicContext dynamicContext = factory.newDynamicContext();
// Register the result resolver with the dynamic context
XResultResolver resultResolver=new AResultResolver(replacementBase);
dynamicContext.setResultResolver(resultResolver);

// Execute the XPath expression
XSequenceCursor cursor = executable.execute(new StreamSource(inputFile), dynamicContext);
```

Registering a collection resolver

You can register implementations of the XCollectionResolver interface with the XDynamicContext.

Procedure

Register a collection resolver with the dynamic context.
The XCollectionResolver implementation registered with the XDynamicContext is used at execution time to retrieve the collection of nodes associated with the URI provided in calls to the fn:collection method. If no collection resolver is registered with the XDynamicContext then calls to fn:collection will result in a recoverable error and the empty sequence is used for the collection.
Note that the collection resolver and the fn:collection function are not meant to resolve document URIs. The source resolver and fn:doc function should be used for this purpose.

Example

The following is a basic example of using a collection resolver.

```
XFactory factory = XFactory.newInstance();

// Prepare the XPath expression
XPathExecutable executable = factory.prepareXPath("count(collection('typeA-typeB'))"

// Register the collection resolver with the dynamic context
XCollectionResolver collectionResolver = new ACollectionResolver(factory);
XDynamicContext dynamicContext = factory.newDynamicContext();
dynamicContext.setCollectionResolver(collectionResolver);

// Execute the XPath expression
XSequenceCursor cursor = executable.execute(dynamicContext);
```

The following is a basic example of an XCollectionResolver implementation.

```
public class ACollectionResolver implements XCollectionResolver {

    private XFactory m_factory;

    public ACollectionResolver(XFactory factory) {
        m_factory = factory;
    }

    public XSequenceCursor getCollection(String uri, String base) {

        // Get the default collection
        if (uri.equals("")) {
            return getCollection("default", base);
        }

        // Get the requested collection
        ArrayList<XItemView> list = new ArrayList<XItemView>();
        StringTokenizer tokenizer = new StringTokenizer(uri, "-");
          XSequenceCursor cursor = null;
          while (tokenizer.hasMoreTokens()) {
              String token = tokenizer.nextToken();
              XSequenceCursor temp = getNodes(new StreamSource("collections.xml"), "/doc/" + token);
```

```
            if (cursor == null) {
                cursor = temp;
            } else {
                cursor = cursor.append(temp);
            }
         }
        return cursor;
    }

     private XSequenceCursor getNodes(Source source, String expression) {
         XPathExecutable executable = m_factory.prepareXPath(expression);
         XSequenceCursor cursor = executable.execute(source);
         return cursor;
     }

}
```

Registering a schema resolver

The XSchemaResolver interface can be implemented and the implementation registered with the XFactory to override the default schema resolution behavior. This includes resolution of imports for schemas registered with XFactory using the registerSchema method and resolving schemas imported in XSLT stylesheets using the xsl:import-schema declaration.

About this task

The default behavior for resolving imports within a schema is to use the base URI of the schema to resolve the imported schema's location. The default behavior for XSLT schema imports is to use the base URI of the xsl:import-schema declaration to resolve the location specified in the declaration.

Procedure

Use the setSchemaResolver method on the XFactory class to register a schema resolver.
The getSchema method returns an instance of the java.util.List interface. This is because the definitions of the schema components for a particular namespace can be split across several distinct schema documents. You can use the getSchema method to return all the schema documents for the particular target namespace associated with all the specified location hints.

Example

The following is a basic example of using a schema resolver.

```
XFactory factory = XFactory.newInstance();

// Set validating to true.
factory.setValidating(true);

// Create the schema resolver and register it with the factory.
factory.setSchemaResolver(new ASchemaResolver(replacementBase));

// Prepare the stylesheet.
XSLTExecutable executable = factory.prepareXSLT(new StreamSource(stylesheetFile));

// Execute the transformation.
Source source = new StreamSource(inputFile);
Result result = new StreamResult(System.out);
executable.execute(source, result);
```

The following is a basic example of an XSchemaResolver implementation.

```
class ASchemaResolver implements XSchemaResolver
{
    String _replacementBase;

    public ASchemaResolver(String replacementBase)
    {
        _replacementBase=replacementBase;
    }

    // Resolve URI, returning the Source that URI represents.
    // Implements the "rebase:" pseudo-scheme.
    public List<? extends Source> getSchema(String namespace, List<String> locations, String baseURI) {
        String rebasePrefix="rebase:";

        List<StreamSource> list = new ArrayList<StreamSource>();
        for (int i = 0; i < locations.size(); i++) {
```

```
            String href = locations.get(i);
            String base = baseURI;
            if(href.startsWith(rebasePrefix)) {
                href=href.substring(rebasePrefix.length());
                base=_replacementBase;
            }

            java.net.URI uri;
            StreamSource source=null;
            try {
                // Get base URI object
                uri = new java.net.URI(base);
                // Resolved relative reference against base URI
                URI resolvedURI = uri.resolve(href);
                // Try to read...
                source = new StreamSource(resolvedURI.toString());
            } catch (java.net.URISyntaxException use) {
                throw new RuntimeException(use);
            }

            list.add(source);
        }
        return list;
    }
}
```

Using an unparsed text resolver

The XUnparsedTextResolver interface can be implemented and the implementation registered with the XDynamicContext to override the default resolution behavior for resources loaded through the XSLT unparsed-text function.

About this task

The default resolution behavior for resources loaded through the XSLT unparsed-text function is to resolve relative URIs based on the base URI from the static context. If the base URI is not available, the current working directory is used. Absolute URIs are used unchanged.

Procedure

Use the setUnparsedTextResolver method on the XDynamicContext interface to register an unparsed text resolver.

Example

The following is a basic example of using an unparsed text resolver.

```
XFactory factory = XFactory.newInstance();

// Prepare the stylesheet.
XSLTExecutable executable = factory.prepareXSLT(new StreamSource(stylesheetFile));

// Create the dynamic context and set the unparsed text resolver.
XDynamicContext dynamicContext = factory.newDynamicContext();
AnUnparsedTextResolver resolver = new AnUnparsedTextResolver(replacementBase);
dynamicContext.setUnparsedTextResolver(resolver);

// Execute the transformation.
Source source = new StreamSource(inputFile);
Result result = new StreamResult(System.out);
executable.execute(source, dynamicContext, result);
```

The following is a basic example of an unparsed text resolver implementation.

```
class AnUnparsedTextResolver implements XUnparsedTextResolver
{
    String _replacementBase;

    public AnUnparsedTextResolver(String replacementBase)
    {
        _replacementBase=replacementBase;
    }

    // Resolve URI, returning the resource that URI represents.
    // Implements the "rebase:" pseudo-scheme.
    public String getResource(String href, String encoding, String base) {
        String rebasePrefix="rebase:";

        if (href.startsWith(rebasePrefix)) {
```

```
            href = href.substring(rebasePrefix.length());
            base = _replacementBase;
        }

        try {
            // Get base URI object
            URI uri = new java.net.URI(base);
            // Resolved relative reference against base URI
            URI resolvedURI = uri.resolve(href);
            // Try to read...
            URL url = resolvedURI.toURL();
            URLConnection urlCon = url.openConnection();
            BufferedInputStream stream = new BufferedInputStream(urlCon.getInputStream());
            StringBuffer buffer = new StringBuffer();
            int s;
            while ((s = stream.read())!= -1) {
                // Do any character manipulation here.
                buffer.append((char)s);
            }
            return buffer.toString();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Using resolvers in a J2EE context

When loading artifacts in a Java 2 Platform, Enterprise Edition (J2EE) context, you should consider the special implications that apply when loading resources from local deployment artifacts.

About this task

Loading local resources—through Class.getResource and Class.getResourceAsStream—from J2EE deployment artifacts such as EARs, WARs, and library JAR files can introduce issues when loading related XML artifacts. Loading an initial local resource using these mechanisms will succeed, but artifacts loaded from the initial resource typically will fail to load without specific consideration.

Here is an example of loading documents at execution time from a stylesheet using the XPath fn:doc function. In this case, the default behavior is to resolve documents based on the base URI from the static context.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

 <xsl:variable name="names" select="doc('names.xml')"/>

 <xsl:template name="loadnames">
  <xsl:copy-of select="$names"/>
 </xsl:template>

</xsl:stylesheet>
```

If this is loaded with the following Java code in a J2EE environment:

```
// create the factory
XFactory factory = XFactory.newInstance();

// load the XSLT file from a local resource within a J2EE deployment artifact
StreamSource source = new StreamSource(XSLTDocFunction.class.getResourceAsStream("/samplexslts/doc.xsl"));

// Create an XSL transform executable
XSLTExecutable xslTransform = factory.prepareXSLT(source);

// Create the result
Result result = new StreamResult(new ByteArrayOutputStream());

// Create a dynamic context specifying the XSLT initial template
XDynamicContext dc = factory.newDynamicContext();
dc.setXSLTInitialTemplate(new QName("loadnames"));

// Execute the transformation
xslTransform.execute(dc, result);
```

you will receive the following error:

```
IXJXE0364W: FATAL ERROR:  IXJXE0774E: [ERR 0693][ERR FODC0005]
The URI string 'names.xml' does not map to an available document.
```

The reason for this error is that in loading the initial XML artifact (doc.xsl), no static content was established for the base URI. In this case, the processor will fall back to looking in the current working directory, which is meaningless in a J2EE environment.

There are three possible ways to fix this situation. Here are the first two:

- Set the baseURI in the static context.

  Adjusting to set the baseURI on the static context would look like this example:

```
// create the factory
XFactory factory = XFactory.newInstance();

// set the baseURI in the static context
URL dataURL = XSLTSchemaAware.class.getResource("/samplexslts/doc.xsl");
XStaticContext staticContext = factory.newStaticContext();
staticContext.setBaseURI(dataURL.toString());

// load the XSLT file from a local resource within a J2EE deployment artifact
StreamSource source = new StreamSource(XSLTDocFunction.class.getResourceAsStream("/samplexslts/doc.xsl"));

// Create an XSL transform executable
XSLTExecutable xslTransform = factory.prepareXSLT(source, staticContext);

// Create the result
Result result = new StreamResult(new ByteArrayOutputStream());

// Create a dynamic context specifying the XSLT initial template
XDynamicContext dc = factory.newDynamicContext();
dc.setXSLTInitialTemplate(new QName("loadnames"));

// Execute the transformation
xslTransform.execute(dc, result);
```

- Load the resource in a way that allows the processor to know the baseURI.

  Adjusting to load the resource in a way that allows the processor to know the baseURI by passing the absolute URL to the StreamSource constructor would look like the following example:

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create the source from a URL
URL dataURL = XSLTSchemaAware.class.getResource("/samplexslts/doc.xsl");
StreamSource source = new StreamSource(dataURL.toString());

// Create an XSL transform executable for the expression
XSLTExecutable xslTransform = factory.prepareXSLT(source);

// Create the result
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Result result = new StreamResult(baos);

// Create a dynamic context specifying the XSLT initial template
XDynamicContext dc = factory.newDynamicContext();
dc.setXSLTInitialTemplate(new QName("loadnames"));

// Execute the transformation
xslTransform.execute(dc, result);
```

These two approaches described will work well when all XML artifacts are in a single J2EE deployment unit, but they will fail if XML artifacts are split across J2EE deployment units because there is no single baseURI for all of the XML artifacts.

To have complete control over XML artifact loading to support cases such as artifacts spread across multiple deployment units, use resolvers to completely control loading behavior.

Procedure

Implement the appropriate resolvers and register those resolvers into the static or dynamic context as appropriate.
These XML artifact loading recommendations apply within a J2EE context for all XML artifacts supported by resolvers such as XML documents (fn:doc(), document()), stylesheets (xsl:include, xsl:import), unparsed text, and XML schemas (XSLT import-schema).

Using a module resolver

The XModuleResolver interface can be implemented and the implementation registered with the XStaticContext to override the default XQuery module resolution behavior. Modules are resolved whenever an XQuery module import is encountered.

About this task

The default module resolution behavior is to attempt to locate one module for each location hint specified in the module import. The default resolution behavior for each location hint is to resolve relative URIs against the base URI from the static context, if the base URI is available, or to interpret them as file paths relative to the current working directory, if the base URI is not available. Absolute URIs are used unchanged. If a module cannot be located for a location hint, the processor ignores it unless no modules can be loaded for the namespace, in which case the processor will emit an error message.

Procedure

Use the setModuleResolver method on the XStaticContext interface to register a module resolver. The getModule method returns an instance of the java.util.List interface. This is because there might be more than one module document for a particular namespace and set of location hints.

Example

The following is a basic example of using a module resolver.

```
XFactory factory = XFactory.newInstance();

// Create the static context
XStaticContext staticContext = factory.newStaticContext();

// Create the module resolver and register it with the static context.
staticContext.setModuleResolver(new AModuleResolver(replacementBase));

// Prepare the query.
XQueryExecutable executable = factory.prepareXQuery(new StreamSource(queryFile), staticContext);

// Execute the transformation.
Source source = new StreamSource(inputFile);
Result result = new StreamResult(System.out);
executable.execute(source, result);
```

The following is a basic example of a module resolver implementation.

```
class AModuleResolver implements XModuleResolver
{
    String _replacementBase;

    public AModuleResolver(String replacementBase)
    {
        _replacementBase=replacementBase;
    }

    // Resolve URI, returning the Source that URI represents.
    // Implements the "rebase:" pseudo-scheme.
    public List<? extends Source> getModule(String namespace, List<String> locations, String baseURI) {
        String rebasePrefix="rebase:";

        List<StreamSource> list = new ArrayList<StreamSource>();
        for (int i = 0; i < locations.size(); i++) {
            String href = locations.get(i);
            String base = baseURI;
            if(href.startsWith(rebasePrefix)) {
                href=href.substring(rebasePrefix.length());
                base=_replacementBase;
```

```
            }

            java.net.URI uri;
            StreamSource source=null;
            try {
                // Get base URI object
                uri = new java.net.URI(base);
                // Resolved relative reference against base URI
                URI resolvedURI = uri.resolve(href);
                // Try to read...
                source = new StreamSource(resolvedURI.toString());
            } catch (java.net.URISyntaxException use) {
                throw new RuntimeException(use);
            }

            list.add(source);
        }
        return list;
    }
}
```

Using external variables and functions

You can use this information to help you to use external variables and functions.

Procedure

- Use external variables.
- Use external functions.

Using external variables

The XML API allows you to access external variables from XPath and XQuery expressions as well a stylesheet parameters from XSLT stylesheets.

About this task

External variables are useful when your XML processing depends on information that is not contained in an input document. If you are writing an application that allows users to search for books by title in a library that is stored as XML, for example, you could prepare a single XQuery expression to find books in the library that uses a variable for the book title to be matched. The resulting XQueryExecutable object can be used for each search that the user submits by binding the submitted book title to the variable. The details of using external variables and stylesheet parameters are presented in separate articles for XPath, XQuery, and XSLT.

Procedure

- Use external variables with XPath.
- Use external variables with XQuery.
- Set parameters with XSLT.

Using external variables with XPath:

When using an XPath expression that uses external variables, supply (or bind) values for each variable using an XDynamicContext instance and optionally declare the types of the variables using an XStaticContext instance.

Procedure

- When preparing an XPath expression that uses external variables, declare the types of the variables using an XStaticContext instance.

This step is optional. If a variable is not declared, the processor assumes its type is item()*. In other words, the value of the variable can be a sequence of any length consisting of items of any type. Declaring a type for your variables can help the processor detect some usage errors statically during preparation.

The XStaticContext interface has two declareVariable methods that each have two parameters—one for the name, and one for the type of the variable. The name is always provided as a QName object, but the type can be a QName or an XSequenceType.

Table 258. XStaticContext declareVariable methods.

The following table explains when to use each form of the declareVariable method.

Method Signature	Purpose
declareVariable(QName name, QName type)	Use when value of the variable is a single atomic value The QName must refer to a built-in type or a global type declared in a schema that has been registered on the XFactory instance used to create the XStaticContext instance. If the QName refers to a non-atomic type, then the processor will treat the variable as having the type element(*, ns:type), where ns:type is the given QName. The XTypeConstants interface has convenient constants available that provide a QName object for each built-in type.
declareVariable(QName name, XSequenceType type)	Use when value of the variable is a single node or a sequence of atomic values or nodes

The following example shows how to prepare an XPath expression that uses variables, with two declared in the static context and one not declared.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a new static context from the factory
XStaticContext staticContext = factory.newStaticContext();

// Define QNames for the names of the variables to be declared
final QName taxRate = new QName("taxRate");
final QName partNumbers = new QName("partNumbers");

// Declare a variable called "taxRate" as an xs:float
staticContext.declareVariable(taxRate, XTypeConstants.FLOAT_QNAME);

// Obtain an XSequenceTypeFactory instance from the factory
XSequenceTypeFactory typeFactory = factory.getSequenceTypeFactory();

// Define a sequence type for a sequence of xs:integer values
XSequenceType integerSequence = typeFactory.atomic(XTypeConstants.INTEGER_QNAME, XSequenceType.OccurrenceIndicator.ZERO_OR_MORE);

// Declare a variable called "partNumbers" as a sequence of xs:integer values
staticContext.declareVariable(partNumbers, integerSequence);

// Create an XPath expression that uses the declared variables, as well as another variable "discount"
// that the processor will assume has type item()*
String expression = "sum(for $partNumber in $partNumbers return /inventory/part[@num=$partNumber]/@price) * (1 - $discount) * (1 + $taxRate)";

// Prepare the XPath expression
XPathExecutable xpath = factory.prepareXPath(expression, staticContext);
```

- To execute an XPath expression that uses external variables, supply (or bind) values for each variable using an XDynamicContext instance.

  An error is raised if you do not supply a value for a variable that is used when executing the XPath expression.

  The XDynamicContext has a number of bind, bindItem, and bindSequence methods. Each has two parameters, where the first is a QName object corresponding to the name of the parameter and the second is the value.

Table 259. XDynamicContext bind, bindItem, and bindSequence methods.

The following table explains when to use each form of the XDynamicContext bind, bindItem, and bindSequence methods.

Method	Purpose
bind	Use when binding a single atomic value There is one form of this method for each of the Java types that is used in the standard mapping of built-in types to Java types. There are two additional forms—one that takes a node and one that takes a source. These are used for binding any node from a DOM tree and parsing a new source to yield a document node, respectively.
bindItem	Use when binding a single item as an XItemView object An XItemView object can be obtained from the result of executing another expression or constructed using an XItemFactory instance.
bindSequence	Use when binding sequences of less than or greater than one item There is one form of this method for each of the Java types that is used in the standard mapping of built-in types to Java types; each accepts an array of values the given type. There is an additional form that takes an XSequenceCursor. An XSequenceCursor can be the result of executing another expression or can be constructed using an XItemFactory instance.

The following example executes the XPath expression prepared in the first example, first binding values for each of the variables it uses.

```
// Create a new dynamic context from the factory
XDynamicContext dynamicContext = factory.newDynamicContext();

// Bind an atomic value for the "taxRate" and "discount" variables
dynamicContext.bind(taxRate, 0.13f);
dynamicContext.bind(new QName("discount"), 0.40);

// Bind a sequence of atomic values for the "partNumbers" variable
dynamicContext.bindSequence(partNumbers, new int[] {2, 1, 2, 3, 2});

// Create an XML input document
String xml = "<inventory>" +
"<part num='1' price='9.99'/>" +
"<part num='2' price='4.47'/>" +
"<part num='3' price='12.99'/>" +
"</inventory>";
StreamSource source = new StreamSource(new StringReader(xml));

// Execute the expression
XSequenceCursor result = xpath.execute(source, dynamicContext);
```

Using external variables with XQuery:

When using an XQuery expression that uses external variables, supply (or bind) values for each variable using an XDynamicContext instance and optionally declare the types of the variables using an XStaticContext instance.

Procedure

- When preparing an XQuery expression that uses external variables, declare the types of the variables using an XStaticContext instance.

  Variables declared in the XStaticContext are only visible to the main module. To make an external variable visible to a library module, it must be declared in the prolog of the library module with an external variable declaration.

  Declaring the variable in the XStaticContext is optional. If a variable is not declared, the processor assumes that its type is item()*. In other words, the value of the variable can be a sequence of any length consisting of items of any type. Declaring a type for your variables can help the processor detect some usage errors statically during preparation.

The XStaticContext interface has two declareVariable methods that each have two parameters—one for the name, and one for the type of the variable. The name is always provided as a QName object, but the type can be a QName or an XSequenceType.

Table 260. XStaticContext declareVariable methods.

The following table explains when to use each form of the declareVariable method.

Method Signature	Purpose
declareVariable(QName name, QName type)	Use when value of the variable is a single atomic value The QName must refer to a built-in type or a global type declared in a schema that has been registered on the XFactory instance used to create the XStaticContext instance. If the QName refers to a non-atomic type, then the processor will treat the variable as having the type element(*, ns:type), where ns:type is the given QName. The XTypeConstants interface has convenient constants available that provide a QName object for each built-in type.
declareVariable(QName name, XSequenceType type)	Use when value of the variable is a single node or a sequence of atomic values or nodes

XQuery also allows variables to be declared in the prolog of an XQuery expression. The following XQuery expression uses two variables, one declared in the prolog and one not.

```
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
declare variable $searchTerms as xs:string+ external;

<table>
    <tr><td>Title</td><td>Author</td></tr>
    {
        for $book in /library/book
        let $value := $book/@*[local-name()=$searchField]
        where exists(for $term in $searchTerms return if (contains($value, $term)) then true() else ())
        return
            <tr>
                <td>{ string($book/@title) }</td>
                <td>{ string($book/@author) }</td>
            </tr>
    }
</table>
```

The variable not declared in the XQuery expression itself can be optionally declared using an XStaticContext instance to provide the processor with type information.

The following example shows how to prepare the above XQuery expression, providing a type for the variable not declared in the XQuery expression itself. It assumes that the expression is accessible using the xquerySource source object.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a new static context from the factory
XStaticContext staticContext = factory.newStaticContext();

// Define a QName for the name of the variable to be declared
final QName searchField = new QName("searchField");

// Declare a variable called "searchField" as an xs:string
staticContext.declareVariable(searchField, XTypeConstants.STRING_QNAME);

// Prepare the XQuery expression
XQueryExecutable xquery = factory.prepareXQuery(xquerySource, staticContext);
```

- To execute an XQuery expression that uses external variables, supply (or bind) values for each variable using an XDynamicContext instance.

  The bindings for external variables will be available to the main module and to any library modules that have an external variable declaration in their prolog for that variable.

  An error is raised if you do not supply a value for a variable that is used when executing the XQuery expression.

  The XDynamicContext has a number of bind, bindItem, and bindSequence methods. Each has two parameters, where the first is a QName object corresponding to the name of the parameter and the second is the value.

Table 261. XDynamicContext bind, bindItem, and bindSequence methods.

The following table explains when to use each form of the XDynamicContext bind, bindItem, and bindSequence methods.

Method	Purpose
bind	Use when binding a single atomic value There is one form of this method for each of the Java types that is used in the standard mapping of built-in types to Java types. There are two additional forms—one that takes a node and one that takes a source. These are used for binding any node from a DOM tree and parsing a new source to yield a document node, respectively.
bindItem	Use when binding a single item as an XItemView object An XItemView object can be obtained from the result of executing another expression or constructed using an XItemFactory instance.
bindSequence	Use when binding sequences of less than or greater than one item There is one form of this method for each of the Java types that is used in the standard mapping of built-in types to Java types; each accepts an array of values the given type. There is an additional form that takes an XSequenceCursor. An XSequenceCursor can be the result of executing another expression or can be constructed using an XItemFactory instance.

The following example executes the XQuery expression prepared in the first example, first binding values for each of the variables it uses.

```
// Create a new dynamic context from the factory
XDynamicContext dynamicContext = factory.newDynamicContext();

// Bind an atomic value for the "searchField" variable
dynamicContext.bind(searchField, "title");

// Bind a sequence of atomic values for the "searchTerms" variable
dynamicContext.bindSequence(new QName("searchTerms"), new String[] {"Lost", "Gables"});

// Create an XML input document
String xml = "<library>" +
"<book title='Lost in the Barrens' author='Farley Mowat'/>" +
"<book title='Anne of Green Gables' author='L. M. Montgomery'/>" +
"</library>";
StreamSource source = new StreamSource(new StringReader(xml));

// Execute the expression
XSequenceCursor result = xquery.execute(source, dynamicContext);
```

Setting parameters with XSLT:

To use parameters in an XSLT stylesheet, declare the parameters as global parameters in the stylesheet itself.

Procedure

- Use the as attribute to declare a type for the parameter.

  The following stylesheet declares two parameters, explicitly giving one of them a type.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                exclude-result-prefixes="xs"
                version="2.0">

  <xsl:param name="targetDate" as="xs:date"/>
  <xsl:param name="window" select="3"/>

  <xsl:template match="/">
    <table>
      <tr><td>Task</td><td>Due</td><td>Status</td></tr>

      <xsl:apply-templates select="todo-list/task[xs:date(@due) le $targetDate + xs:dayTimeDuration(concat('P', $window, 'D'))]">
          <xsl:sort select="@due"/>
      </xsl:apply-templates>
    </table>
  </xsl:template>
```

```
  <xsl:template match="task">
    <tr>
      <td><xsl:value-of select="."/></td>
      <td><xsl:value-of select="format-date(xs:date(@due), '[MNn] [D1o]')"/></td>
      <td>
          <xsl:choose>
          <xsl:when test="xs:date(@due) lt $targetDate">OVERDUE by
             <xsl:value-of select="days-from-duration($targetDate - xs:date(@due))"/> day(s)</xsl:when>
          <xsl:otherwise>Due in <xsl:value-of select="days-from-duration(xs:date(@due) - $targetDate)"/> day(s)</xsl:otherwise>
          </xsl:choose>
      </td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

- Prepare an XSLT stylesheet with global parameters in exactly the same way as any other stylesheet because there is no need to declare anything in the static context.

  Assuming that the stylesheet above is available using the xsltSource Source object, the code below prepares the stylesheet.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Prepare the XSLT stylesheet
XSLTExecutable xslt = factory.prepareXSLT(xsltSource);
```

- To execute an XSLT stylesheet that uses parameters, supply (or bind) values for each parameter using an XDynamicContext instance.

  You must supply a value for any required parameters or an error is raised. For other parameters, if you do not supply a value, a default value is used—either one supplied in the stylesheet or a zero-length string.

  The XDynamicContext has a number of bind, bindItem, and bindSequence methods. Each has two parameters, where the first is a QName object corresponding to the name of the parameter and the second is the value.

Table 262. XDynamicContext bind, bindItem, and bindSequence methods.

The following table explains when to use each form of the XDynamicContext bind, bindItem, and bindSequence methods.

Method	Purpose
bind	Use when binding a single atomic value There is one form of this method for each of the Java types that is used in the standard mapping of built-in types to Java types. There are two additional forms—one that takes a Node object and one that takes a Source object. These are used for binding any node from a DOM tree and parsing a new source to yield a document node, respectively.
bindItem	Use when binding a single item as an XItemView object An XItemView object can be obtained from the result of executing another expression or constructed using an XItemFactory instance.
bindSequence	Use when binding sequences of less than or greater than one item There is one form of this method for each of the Java types that is used in the standard mapping of built-in types to Java types; each accepts an array of values the given type. There is an additional form that takes an XSequenceCursor. An XSequenceCursor can be the result of executing another expression or can be constructed using an XItemFactory instance.

  The following example executes the XSLT stylesheet prepared in the first example, first binding values for each of the parameters it will use. It assumes that a Result object called xsltResult has already been created.

```
// Create an xs:date value for the "targetDate" parameter with date "April 10, 2009"
XMLGregorianCalendar date = DatatypeFactory.newInstance().newXMLGregorianCalendarDate(2009, 4, 10, DatatypeConstants.FIELD_UNDEFINED);

// Create a new dynamic context from the factory
XDynamicContext dynamicContext = factory.newDynamicContext();

// Bind an atomic value for the "targetDate" parameter
dynamicContext.bind(new QName("targetDate"), date);

// Bind an atomic value for the "window" parameter
dynamicContext.bind(new QName("window"), 7);
```

```
// Create an XML input document
String xml = "<todo-list>" +
"<task due='2009-03-31' completed=''>File Quarterly Report</task>" +
"<task due='2009-05-04' completed='2009-04-22'>Review candidate resumes</task> +
"<task due='2009-04-16' completed=''>Order stock</task>" +
"<task due='2009-05-01' completed=''>Buy concert tickets</task>" +
"</todo-list>";
StreamSource source = new StreamSource(new StringReader(xml));

// Execute the stylesheet
xslt.execute(source, dynamicContext, xsltResult);
```

Using external functions

The XML API allows you to use external functions with XPath, XQuery, and XSLT.

About this task

You can define external functions when you want to use operations that are difficult or impossible to express directly by core functions defined in functions and operators, additional functions defined in the XSLT specification, constructor functions named after an atomic type, stylesheet functions defined using xsl:function declaration of XSLT, or user-defined functions in XQuery. Please note that external functions cannot be used to override built-in functions. For example, suppose you are using an external function, my:power(arg1 as xs:double, arg2 as xs:double), to calculate the value of the first argument raised to the power of the second argument; you could write a Java method taking two Java primitive double arguments that performs the calculation. The details of using external functions are presented in separate articles.

Procedure

- Use external functions with XPath.
- Use external functions with XQuery.
- Use external functions with XSLT.

Using external functions with XPath:

When using an XPath expression that uses external functions, declare the function signatures using an XStaticContext instance and supply (or bind) a Java implementation for each function using an XDynamicContext instance.

Procedure

1. When preparing an XPath expression that uses external functions, declare the function signatures using an XStaticContext instance.

   The XStaticContext interface has two declareFunction methods that each have three parameters—one for the name, one for the return type of the function, and an array for the types of the arguments. The name is always provided as a QName object, but the types can be QNames or XSequenceTypes. The function name, return type, and argument types must uniquely identify the function.

Table 263. XStaticContext declareFunction methods.

This table explains when to use each form of the declareFunction method.

Method Signature	Purpose
declareFunction(QName name, QName type, QName[] argTypes)	Use when the return value and arguments of the function are all single atomic values The type QNames must refer to built-in types or global types declared in a schema that has been registered on the XFactory instance used to create the XStaticContext instance. If a QName refers to a non-atomic type, the processor will treat it as the type element(*, ns:type), where ns:type is the given QName. The XTypeConstants interface has convenient constants available that provide a QName object for each built-in type.
declareFunction(QName name, XSequenceType type, XSequenceType[] argTypes)	Use when any of the arguments or the return value of the function is a node or a sequence of atomic values or nodes

The following example shows how to prepare an XPath expression that uses an external function.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a new static context
XStaticContext staticContext = factory.newStaticContext();

// Declare a namespace for the function
staticContext.declareNamespace("my", "http://myfunc");

// Create a QName for the name of the function
QName methodQName = new QName("http://myfunc", "pow");

// Declare the function on the static context
staticContext.declareFunction(methodQName, XTypeConstants.DOUBLE_QNAME, new QName[]{XTypeConstants.DOUBLE_QNAME, XTypeConstants.DOUBLE_QNAME});

// Create an XPath executable for the expression
XPathExecutable executable = factory.prepareXPath("sum(/polynomial/term/(my:pow(2, @power) * @coefficient))", staticContext);
```

2. To execute an XPath expression that uses external functions, supply (or bind) the Java methods that implement the functions using an XDynamicContext instance.

   Use Java reflection to obtain a java.lang.reflect.Method object for the function. If the method is an instance method, an instance object is required when binding this function.

   An error is raised if you do not supply a Java method for a function that is used when executing the XPath expression.

   The XDynamicContext has two bindFunction methods. Each requires a QName object corresponding to the name of the function and a Method object identifying the Java method that will provide the implementation for the function.

Table 264. XDynamicContext bindFunction methods.

This table explains when to use each form of the XDynamicContext bindFunction methods.

Method Name	Purpose
bindFunction(QName qname, Method method)	Use when binding a static method
bindFunction(QName qname, Method method, Object instanceObject)	Use when binding an instance method

The following example executes the XPath expression prepared in the first example, first binding a method for the function it uses. In this example, the static pow(double a, double b) method of the java.lang.Math class is used to provide the implementation for the external function.

```
// Create a new dynamic context
XDynamicContext dynamicContext = factory.newDynamicContext();

// Retrieve the java.lang.reflect.Method object for this function
Method method = Math.class.getMethod("pow", Double.TYPE, Double.TYPE);

// Bind the function to the dynamic context
dynamicContext.bindFunction(methodQName, method);

// Create an XML input document
String xml = "<polynomial>" +
"<term power='2' coefficient='3'/>" +
"<term power='1' coefficient='-2'/>" +
"<term power='0' coefficient='1'/>" +
"</polynomial>";
StreamSource source = new StreamSource(new StringReader(xml));

// Execute the expression
XSequenceCursor result = executable.execute(source, dynamicContext);


// Serialize the result to System.out
result.exportItem(new StreamResult(System.out));
```

Using external functions with XQuery:

When using an XQuery expression that uses external functions, declare the function signatures either in the XQuery prolog as external functions or using an XStaticContext instance. Supply (or bind) a Java implementation for each function using an XDynamicContext instance.

Procedure

1. When preparing an XQuery expression that uses external functions, declare the function signatures in the XQuery prolog or using an XStaticContext instance.

   Functions declared in the XStaticContext are only visible to the main module. For an external function to be visible to a library module, it must be declared in the prolog of that library module.

   The XStaticContext interface has two declareFunction methods that each have three parameters—one for the name, one for the return type of the function, and an array for the types of the arguments. The name is always provided as a QName object, but the types can be QNames or XSequenceTypes. The function name, return type, and argument types must uniquely identify the function.

Table 265. XStaticContext declareFunction methods.

This table explains when to use each form of the declareFunction method.

Method Signature	Purpose
declareFunction(QName name, QName type, QName[] argTypes)	Use when the return value and arguments of the function are all single atomic values The type QNames must refer to built-in types or global types declared in a schema that has been registered on the XFactory instance used to create the XStaticContext instance. If a QName refers to a non-atomic type, the processor will treat it as the type element(*, ns:type), where ns:type is the given QName. The XTypeConstants interface has convenient constants available that provide a QName object for each built-in type.
declareFunction(QName name, XSequenceType type, XSequenceType[] argTypes)	Use when any of the arguments or the return value of the function is a node or a sequence of atomic values or nodes

   The following XQuery expression uses three functions, two of which are declared in the prolog.

```
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
declare namespace trig = "http://www.example.org/trigonometry";
declare function trig:arctan($ratio as xs:double) as xs:double external;
declare function trig:sin($angle as xs:double) as xs:double external;

<ramps>
{
    for $ramp in ramps/ramp
    let $angleRadians := trig:arctan($ramp/height div $ramp/base)
    let $angle := trig:toDegrees($angleRadians)
    let $length := $ramp/height div trig:sin($angleRadians)
    return
        element ramp
        {
            $ramp/height,
            $ramp/base,
            element angle { $angle },
            element length { $length }
        }
}
</ramps>
```

   Assuming that the query above is available using the xquerySource Source object, the code below prepares the query. The function that was not declared in the query itself, trig:toDegrees, is declared on the XStaticContext instance.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a new static context
XStaticContext staticContext = factory.newStaticContext();

// Declare a namespace for the functions
staticContext.declareNamespace("trig", "http://www.example.org/trigonometry");

// Create a QName for the trig:toDegrees function
QName toDegreesQName = new QName("http://www.example.org/trigonometry", "toDegrees");

// Declare the function on the static context
staticContext.declareFunction(toDegreesQName, XTypeConstants.DOUBLE_QNAME, new QName[]{XTypeConstants.DOUBLE_QNAME});

// Create an XQuery executable for the query
XQueryExecutable executable = factory.prepareXQuery(xquerySource, staticContext);
```

2. To execute an XQuery expression that uses external functions, supply (or bind) the Java methods that implement the functions using an XDynamicContext instance.

The bindings for external functions will be available to the main module and to any library modules that have an external function declaration in their prolog for that function.

Use Java reflection to obtain a java.lang.reflect.Method object for the function. If the method is an instance method, an instance object is required when binding this function.

An error is raised if you do not supply a Java method for a function that is used when executing the XQuery expression.

The XDynamicContext has two bindFunction methods. Each requires a QName object corresponding to the name of the function and a Method object identifying the Java method that will provide the implementation for the function.

Table 266. XDynamicContext bindFunction methods.

This table explains when to use each form of the XDynamicContext bindFunction methods.

Method Name	Purpose
bindFunction(QName qname, Method method)	Use when binding a static method
bindFunction(QName qname, Method method, Object instanceObject)	Use when binding an instance method

The following example executes the XQuery expression prepared in the first example, first binding methods for the functions it uses. In this example, the static atan, sin, and toDegrees methods of the java.lang.Math class are used to provide the implementations for the external functions.

```
// Create a new dynamic context
XDynamicContext dynamicContext = factory.newDynamicContext();

// Retrieve the java.lang.reflect.Method object for the trig:toDegrees function
Method toDegreesMethod = Math.class.getMethod("toDegrees", Double.TYPE);

// Bind the function to the dynamic context
dynamicContext.bindFunction(toDegreesQName, toDegreesMethod);

// Create QNames for the trig:arctan and trig:sin functions
QName arctanQName = new QName("http://www.example.org/trigonometry", "arctan");
QName sinQName = new QName("http://www.example.org/trigonometry", "sin");

// Retrieve the java.lang.reflect.Method objects for the trig:arctan and trig:sin functions
// then bind them to the dynamic context
Method arctanMethod = Math.class.getMethod("atan", Double.TYPE);
Method sinMethod = Math.class.getMethod("sin", Double.TYPE);
dynamicContext.bindFunction(arctanQName, arctanMethod);
dynamicContext.bindFunction(sinQName, sinMethod);

// Create an XML input document
String xml = "<ramps>" +
"<ramp><base>4</base><height>4</height></ramp>" +
"<ramp><base>4</base><height>3</height></ramp>" +
"<ramp><base>10</base><height>2</height></ramp>" +
"</ramps>";
StreamSource source = new StreamSource(new StringReader(xml));

// Execute the query
XSequenceCursor result = executable.execute(source, dynamicContext);

// Serialize the result to System.out
result.exportItem(new StreamResult(System.out));
```

Using query-declared external functions with XQuery:

As an alternative to binding Java methods to functions in a query using the API, Java external functions can be declared directly within a query. The only additional configuration required is for bound Java classes to exist on the classpath during query execution.

Procedure

Using the java-extension XQuery option declaration, you can bind a prefix to a Java class.

```
declare option xltxe:java-extension "prefix = className";
```

Note: Any prefix name can be used for the java-extension element as long as it is bound to the `http://www.ibm.com/xmlns/prod/xltxe-j` namespace.

After binding a prefix to a Java class, methods within the bound class can be invoked by specifying the prefix and method name separated by a colon:

prefix:methodName(Params*)

Example

Invoking Static Methods

When preparing an XQuery source that uses query-declared external functions, declare the prefix to Java class binding:

```
declare namespace calc="http://com.example/myApp/Calculator";
declare namespace sf="http://com.example/myApp/standardFormat";

declare namespace xltxe="http://www.ibm.com/xmlns/prod/xltxe-j";
declare option xltxe:java-extension "calc = org.company.Calculator";
declare option xltxe:java-extension "sf = org.standards.Formatter";

sf:format(calc:sqrt(64), "ISO-42.7")
```

Assuming that this query is available through the xquerySource Source object, the following code prepares the query:

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create an XQuery executable for the query
XQueryExecutable executable = factory.prepareXQuery(xquerySource);
```

The following code executes the query that was prepared in the example:

```
// Create a result object to store the transformation result
Result result = new StreamResult(System.out);

// Execute the XSLT executable
XSequenceCursor xsc = executable.execute();

// Output the result
xsc.exportSequence(res);
```

The example query provided assumes that the org.company.Calculator class contains a static method sqrt() that takes one parameter and the org.standards.Formatter class contains a static method format() that takes two parameters. At prepare time, the classes are not required on the classpath; but they are required during execution of the query.

The following are example implementations of the org.company.Calculator and org.standards.Formatter classes:

```
package org.company;

public class Calculator {
    public static int sqrt(int val) {
        return (int)Math.sqrt(val);
    }
```

```
package org.standards;

public class Formatter {
    public static String format(int val, String pattern) {
        return "Formatting " + val + " using pattern " + pattern;
    }
}
```

Invoking Instance Methods

Invoking instance methods from a class is slightly different from invoking static methods because of the requirement for an instance object. In order to obtain an instance object from a Java class within a query, you must invoke its new constructor:

```
prefix:new(Params*)
```

You can then store the result in an XQuery variable declaration as demonstrated by the following query:

```
declare namespace car="http://com.example/myApp/car";
declare namespace xltxe="http://www.ibm.com/xmlns/prod/xltxe-j";
declare option xltxe:java-extension "car = org.automobile.Car";
declare variable $var := car:new(3);

car:getDoors($var)
```

Assuming that this query is available through the xquerySource Source object, the following code prepares the query.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create an XQuery executable for the query
XQueryExecutable executable = factory.prepareXQuery(xquerySource);
```

The following code executes the query prepared in the example:

```
// Create a result object to store the transformation result
Result result = new StreamResult(System.out);

// Execute the XSLT executable
XSequenceCursor xsc = executable.execute();

// Output the result
xsc.exportSequence(res);
```

The example query assumes that org.automobile.Car class contains a constructor that takes an argument of type int. In addition, the org.automobile.Car class also contains an instance method getDoors() that takes no arguments. The syntax for invoking instance methods from query-declared external functions require that the created instance object be passed in as the first argument.

The following is an example implementation of the org.automobile.Car class:

```
package org.automobile;

public class Car {
    private int doors;

    public Car (int doors) {
        this.doors = doors;
    }

    public int getDoors() {
        return doors;
    }
}
```

Inheritance with instance methods is also supported. If the org.automobile.Car class has a subclass org.automobile.Sedan, you can create an instance of the org.automobile.Sedan class and use it to call methods in org.automobile.Car. This is demonstrated by the following query:

```
declare namespace car="http://com.example/myApp/car";
declare namespace sedan="http://com.example/myApp/sedan";
declare namespace xltxe="http://www.ibm.com/xmlns/prod/xltxe-j";
declare option xltxe:java-extension "car = org.automobile.Car";
```

```
declare option xltxe:java-extension "sedan = org.automobile.Sedan";

declare variable $var := sedan:new(5);

car:getDoors($var)
```

The following is an example implementation for org.automobile.Sedan:

```
package org.automobile;

public class Sedan extends Car {
    public Sedan (int doors) {
        super(doors);
    }
}
```

Limitation: The mechanism used for resolving methods in a Java class requires that only one method exists, matching in name and arity. If multiple methods exist with the same name and different arity, an error is thrown.

Using external functions with XSLT:

When using an XSLT stylesheet that uses external functions, declare the function signatures using an XStaticContext instance and supply (or bind) a Java implementation for each function using an XDynamicContext instance.

Procedure

1. When preparing an XSLT stylesheet that uses external functions, declare the function signatures using an XStaticContext instance.

   The XStaticContext interface has two declareFunction methods that each have three parameters—one for the name, one for the return type of the function, and an array for the types of the arguments. The name is always provided as a QName object, but the types can be QNames or XSequenceTypes. The function name, return type, and argument types must uniquely identify the function.

   A stylesheet function could have the same name as an external function. If the stylesheet function has an override attribute with the value of yes, any reference to that function name in the stylesheet is a reference to the stylesheet function; if it has an override attribute with the value of no, it is a reference to the external function.

Table 267. XStaticContext declareFunction methods.

This table explains when to use each form of the declareFunction method.

Method Signature	Purpose
declareFunction(QName name, QName type, QName[] argTypes)	Use when the return value and arguments of the function are all single atomic values The type QNames must refer to built-in types or global types declared in a schema that has been registered on the XFactory instance used to create the XStaticContext instance. If a QName refers to a non-atomic type, then the processor will treat it as the type element(*, ns:type), where ns:type is the given QName. The XTypeConstants interface has convenient constants available that provide a QName object for each built-in type.
declareFunction(QName name, XSequenceType type, XSequenceType[] argTypes)	Use when any of the arguments or the return value of the function is a node or a sequence of atomic values or nodes

The following stylesheet uses an external function referred to by the QName ext:pow.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ext="http://www.example.com/functions"
                version="2.0">

    <xsl:output method="text"/>

    <xsl:template match="polynomial">
        <xsl:variable name="p" select="."/>
```

```
        <xsl:for-each select="1 to 5">
            <xsl:text>x = </xsl:text>
            <xsl:value-of select="."/>
            <xsl:text>; </xsl:text>
            <xsl:value-of select="for $t in $p/term return concat($t/@coefficient, 'x^', $t/@power)"
                          separator=" + "/>
            <xsl:text> = </xsl:text>
            <xsl:value-of select="sum($p/term/(ext:pow(current(), @power) * @coefficient))"/>
            <xsl:text>&#xA;</xsl:text>
        </xsl:for-each>
    </xsl:template>

</xsl:stylesheet>
```

Assuming that the stylesheet above is available using the xsltSource Source object, the code below prepares the stylesheet.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create a new static context
XStaticContext staticContext = factory.newStaticContext();

// Declare a namespace for the function
staticContext.declareNamespace("ext", "http://www.example.com/functions");

// Create a QName for the name of the function
QName methodQName = new QName("http://www.example.com/functions", "pow");

// Declare the function on the static context
staticContext.declareFunction(methodQName, XTypeConstants.DOUBLE_QNAME, new QName[]{XTypeConstants.DOUBLE_QNAME, XTypeConstants.DOUBLE_QNAME});

// Create an XSLT executable for the stylesheet
XSLTExecutable executable = factory.prepareXSLT(xsltSource, staticContext);
```

2. To execute an XSLT stylesheet that uses external functions, supply (or bind) the Java methods that implement the functions using an XDynamicContext instance.

   Use Java reflection to obtain a java.lang.reflect.Method object for the function. If the method is an instance method, an instance object is required when binding this function.

   An error is raised if you do not supply a Java method for a function that is used when executing the XSLT stylesheet.

   The XDynamicContext has two bindFunction methods. Each requires a QName object corresponding to the name of the function and a Method object identifying the Java method that will provide the implementation for the function.

Table 268. XDynamicContext bindFunction methods.

This table explains when to use each form of the XDynamicContext bindFunction methods.

Method Name	Purpose
bindFunction(QName qname, Method method)	Use when binding a static method
bindFunction(QName qname, Method method, Object instanceObject)	Use when binding an instance method

The following example executes the XSLT stylesheet prepared in the first example, first binding a method for the function it uses. In this example, the static pow(double a, double b) method of the java.lang.Math class is used to provide the implementation for the external function.

```
// Create a new dynamic context
XDynamicContext dynamicContext = factory.newDynamicContext();

// Retrieve the java.lang.reflect.Method object for this function
Method method = Math.class.getMethod("pow", Double.TYPE, Double.TYPE);

// Bind the function to the dynamic context
dynamicContext.bindFunction(methodQName, method);

// Create an XML input document
String xml = "<polynomial>" +
"<term power='2' coefficient='3'/>" +
"<term power='1' coefficient='-2'/>" +
"<term power='0' coefficient='1'/>" +
"</polynomial>";
StreamSource source = new StreamSource(new StringReader(xml));

// Execute the stylesheet
XSequenceCursor result = executable.execute(source, dynamicContext);
```

```
// Serialize the result to System.out
result.exportItem(new StreamResult(System.out), executable.getOutputParameters());
```

Using stylesheet-declared external functions with XSLT:

As an alternative to binding Java methods to functions in a stylesheet using the API, Java external functions can be declared directly within a stylesheet. The only additional configuration required is for bound Java classes to exist on the classpath during stylesheet execution.

Procedure

Using the java-extension element, you can bind a prefix to a Java class.

```
<xltxe:java-extension
prefix = string
class = string />
```

Note: Any prefix name can be used for the java-extension element as long as it is bound to the `http://www.ibm.com/xmlns/prod/xltxe-j` namespace.

After binding a prefix to a Java class, methods within the bound class can be invoked by specifying the prefix and method name separated by a colon:

prefix:methodName(Params*)

Example

Invoking Static Methods

When preparing an XSLT stylesheet that uses stylesheet-declared external functions, declare the prefix to Java class binding:

```
<xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xltxe="http://www.ibm.com/xmlns/prod/xltxe-j"
     xmlns:calc="http://com.example/myApp/calculator"
     xmlns:sf="http://com.example/myApp/standardFormat">

<xltxe:java-extension prefix="calc" class="org.company.Calculator"/>
<xltxe:java-extension prefix="sf" class="org.standards.Formatter"/>

<xsl:template match="/">
     <xsl:value-of select="sf:format(calc:sqrt(64), 'ISO-42.7')"/>
</xsl:template>

</xsl:stylesheet>
```

Assuming that this stylesheet is available through the xsltSource Source object, the following code prepares the stylesheet:

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create an XSLT executable for the stylesheet
XSLTExecutable executable = factory.prepareXSLT(xsltSource);
```

The following code executes the stylesheet that was prepared in the example:

```
// Create the xml input
String xml = "<doc/>";

// Create a result object to store the transformation result
Result result = new StreamResult(System.out);

// Execute the XSLT executable
executable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())), result);
```

The example stylesheet provided assumes that the org.company.Calculator class contains a static method sqrt() that takes one parameter and the org.standards.Formatter class contains a static method format() that takes two parameters. At prepare time, the classes are not required on the classpath; but they are required during execution of the stylesheet.

The following are example implementations of the org.company.Calculator and org.standards.Formatter classes:

```
package org.company;

public class Calculator {
    public static int sqrt(int val) {
        return (int)Math.sqrt(val);
    }
}
```

```
package org.standards;

public class Formatter {
    public static String format(int val, String pattern) {
        return "Formatting " + val + " using pattern " + pattern;
    }
}
```

Invoking Instance Methods

Invoking instance methods from a class is slightly different from invoking static methods because of the requirement for an instance object. In order to obtain an instance object from a Java class within a stylesheet, you must invoke its new constructor:

```
prefix:new(Params*)
```

You can then store the result in an xsl:variable element as demonstrated by the following stylesheet:

```
<xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xltxe="http://www.ibm.com/xmlns/prod/xltxe-j"
     xmlns:car="http://com.example/myApp/car">

<xltxe:java-extension prefix="car" class="org.automobile.Car"/>

<xsl:variable name="var" select="car:new(3)"/>

<xsl:template match="/">
     <xsl:value-of select="car:getDoors($var)"/>
</xsl:template>

</xsl:stylesheet>
```

Assuming that this stylesheet is available through the xsltSource Source object, the following code prepares the stylesheet.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Create an XSLT executable for the stylesheet
XSLTExecutable executable = factory.prepareXSLT(xsltSource);
```

The following code executes the stylesheet prepared in the example:

```
// Create the xml input
String xml = "<doc/>";

// Create a result object to store the transformation result
Result result = new StreamResult(System.out);
```

```
// Execute the XSLT executable
executable.execute(new StreamSource(new ByteArrayInputStream(xml.getBytes())), result);
```

The example stylesheet assumes that the org.automobile.Car class contains a constructor that takes an argument of type int. In addition, the org.automobile.Car class also contains an instance method getDoors() that takes no arguments. The syntax for invoking instance methods from stylesheet-declared external functions require that the created instance object be passed in as the first argument.

The following is an example implementation of the org.automobile.Car class:

```
package org.automobile;

public class Car {
    private int doors;

    public Car (int doors) {
        this.doors = doors;
    }

    public int getDoors() {
        return doors;
    }
}
```

Inheritance with instance methods is also supported. If the org.automobile.Car class has a subclass org.automobile.Sedan, you can create an instance of the org.automobile.Sedan class and use it to call methods in org.automobile.Car. This is demonstrated by the following stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xltxe="http://www.ibm.com/xmlns/prod/xltxe-j"
     xmlns:sedan="http://com.example/myApp/sedan"
     xmlns:car="http://com.example/myApp/car">

<xltxe:java-extension prefix="sedan" class="org.automobile.Sedan"/>
<xltxe:java-extension prefix="car" class="org.automobile.Car"/>

<xsl:variable name="var" select="sedan:new(5)"/>

<xsl:template match="/">
     <xsl:value-of select="car:getDoors($var)"/>
</xsl:template>

</xsl:stylesheet>
```

The following is an example implementation for org.automobile.Sedan:

```
package org.automobile;

public class Sedan extends Car {
    public Sedan (int doors) {
        super(doors);
    }
}
```

Limitation: The mechanism used for resolving methods in a Java class requires that only one method exists, matching in name and arity. If multiple methods exist with the same name and different arity, an error is thrown.

Creating items and sequences

You can use this information to help you to create items and sequences using the XItemFactory as well as to use sequence types.

Procedure

- Create items and sequences using the XItemFactory.
- Use sequence types.

Creating items and sequences using the XItemFactory

You can use XItemFactory to create new items and sequences of items of different types.

Procedure

- Use the getItemFactory method in the XFactory class to create instances of XItemFactory and call the appropriate item method to create an item of a specific type.

  An item itself can be a node or an atomic value such as an integer, string, or boolean.

  The following is an example of using the XItemFactory to create new items of different types.

```
// Create an XFactory
XFactory factory = XFactory.newInstance();

// Create an XItemFactory
XItemFactory itemFactory = factory.getItemFactory();

// Create a new atomic item of a type which is the default mapping of the xs:string built-in type to java.lang.String
XItemView stringItem = itemFactory.item("Lets see");

// Create a new atomic item of type int
XItemView intItem = itemFactory.item(3, XTypeConstants.INT_QNAME);

// Create a new atomic item of type boolean
boolean boolValue = false;
XItemView booleanItem = itemFactory.item(boolValue, XTypeConstants.BOOLEAN_QNAME);

 // Create Node type
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
// Parse the input file
Document doc = db.parse(INPUT_File);
Node node = doc.getFirstChild();
XItemView item =itemFactory.item(node)

 // Create a new item of complex type from a Source
StreamSource source = new StreamSource(INPUT_File);
XItemView complexItem = itemFactory.item(source);
```

  As shown in the example, you can use item method with one argument as the value of the method and the type is evaluated based on the mapping rules between built-in types and Java types.

- Use the getItemFactory method in the XFactory class to create instances of XItemFactory and call the appropriate sequence method to create an XSequenceCursor that represents a sequence of items providing cursor access to the items in the sequence.

  The following is an example of using XItemFactory to create a homogeneous sequence.

```
// Create an XFactory
XFactory factory = XFactory.newInstance();

// Create an XItemXFactory
XItemFactory itemFactory =xfactory.getItemFactory();

//Create a sequence of int values
XSequenceCursor intSeq = xfactory.sequence(new int[]{1,2,3});

//Create a sequence of String values
XSequenceCursor stringSeq = xfactory.sequence(new String[]{"This", "is", "a", "test"},XTypeConstants.STRING_QNAME );
```

  The following is an example of using the XItemFactory to create a sequence of items of different types.

```
// Create an XFactory
XFactory factory = XFactory.newInstance();

// Create an XItemXFactory
XItemFactory itemFactory = factory.getItemFactory();

//Create an Array of the newly created items
XItemView[] items = new XItemView[2];
items[0] =itemFactory.item(boolValue, XTypeConstants.BOOLEAN_QNAME);
items[1] = itemFactory.item(intValue, XTypeConstants.INT_QNAME);

// Create a sequence of items
XSequenceCursor seq = itemFactory.sequence(items);
```

Using sequence types

You can use sequence types when declaring variables for the return or parameter types of functions.

About this task

In the XML API, sequence types are represented using XSequenceType objects. XSequenceType objects can represent the same set of sequence types as the SequenceType syntax defined in XPath 2.0. XSequenceType objects are created through the XSequenceTypeFactory interface. You can obtain an instance of XSequenceTypeFactory using the getSequenceTypeFactory() method on an XFactory instance.

Procedure

- XSequenceTypeFactory has several methods for creating XSequenceType instances.

  Each method on the XSequenceTypeFactory interface has an XSequenceType.OccurrenceIndicator parameter except for emptySequence(). OccurrenceIndicator is an enum that represents the cardinality of the sequence; ZERO_OR_ONE corresponds to "?" in SequenceType syntax, ZERO_OR_MORE to "*", ONE_OR_MORE to "+", and ONE to no occurrence indicator for sequences of exactly one item.
- The XSequenceTypeFactory interface methods use QNames to refer to the names of nodes such as elements and attributes, and also to types and global element and attribute declarations.

  QNames representing types must refer to built-in types or to types defined in schemas that have been registered with the same XFactory instance used to obtain the XSequenceTypeFactory instance. Similarly, QNames representing global element or attribute declarations must refer to global declarations present in schemas that have been registered with the same XFactory instance.

Example

Table 269. XSequenceTypeFactory methods and examples of sequence types. The following table lists each method of the XSequenceTypeFactory and gives examples of the sequence types (using SequenceType syntax) that each can create.

Method Signature	Example Sequence Types	Comments
emptySequence()	empty-sequence()	
item(OccurrenceIndicator cardinality)	item()*	
atomic(QName typeName, OccurrenceIndicator cardinality)	xs:integer	"sc:type" is a QName referring to a user-defined schema type.
	sc:type+	
documentNode(OccurrenceIndicator cardinality)	document-node()?	
documentNodeWithElement(QName elementNameOrWildcard, QName typeName, boolean nillable, OccurrenceIndicator cardinality)	document-node(element())	"ns:elem" is a QName representing an element name, and "sc:type" is a QName referring to a user-defined schema type. The elementNameOrWildcard and typeName parameters are optional; use null if the element name or type does not matter.
	document-node(element(ns:elem))?	
	document-node(element(*, sc:type?))	
documentNodeWithSchemaElement(QName elementName, OccurrenceIndicator cardinality)	document-node(schema-element(sc:elemDecl))	"sc:elemDecl" is a QName referring to a global element declaration in a schema.
element(QName elementNameOrWildcard, QName typeName, boolean nillable, OccurrenceIndicator cardinality)	element(*)	"ns:elem" is a QName representing an element name, and "sc:type" is a QName referring to a user-defined schema type. The elementNameOrWildcard and typeName parameters are optional; use null if the element name or type does not matter.
	element(ns:elem, sc:type)*	
attribute(QName attributeNameOrWildcard, QName typeName, OccurrenceIndicator cardinality)	attribute()+	"ns:attrib" is a QName representing an attribute name. The attributeNameOrWildcard and typeName parameters are optional; use null if the attribute name or type does not matter.
	attribute(ns:attrib)	
	attribute(ns:attrib, xs:string)?	
schemaElement(QName elementName, OccurrenceIndicator cardinality)	schema-element(sc:elemDecl)*	"sc:elemDecl" is a QName referring to a global element declaration in a schema.
schemaAttribute(QName attributeName, OccurrenceIndicator cardinality)	schema-attribute(sc:attribDecl)	"sc:attribDecl" is a QName referring to a global attribute declaration in a schema.

Table 269. XSequenceTypeFactory methods and examples of sequence types (continued). The following table lists each method of the XSequenceTypeFactory and gives examples of the sequence types (using SequenceType syntax) that each can create.

Method Signature	Example Sequence Types	Comments
processingInstruction(QName piNCName, OccurrenceIndicator cardinality)	processing-instruction()?	"pinst" is an NCName representing the name of a processing instruction.
	processing-instruction("pinst")	
	processing-instruction(pinst)	
comment(OccurrenceIndicator cardinality)	comment()	
text(OccurrenceIndicator cardinality)	text()+	
node(OccurrenceIndicator cardinality)	node()*	

The following example shows how to create XSequenceType objects to represent various sequence types. Some of the results refer to types and declarations from a schema. The example assumes that the schema is available through the schemaSource source object.

```
// Create the factory
XFactory factory = XFactory.newInstance();

// Obtain an XSeqeuenceTypeFactory instance
XSequenceTypeFactory stFactory = factory.getSequenceTypeFactory();

// Create a sequence type for a sequence of xs:integer values: "xs:integer*"
XSequenceType integerSequenceType = stFactory.atomic(
        XTypeConstants.INTEGER_QNAME,
        XSequenceType.OccurrenceIndicator.ZERO_OR_MORE);

// Create a sequence type for a single node: "node()"
XSequenceType nodeType = stFactory.node(OccurrenceIndicator.ONE);

// Define a constant for the target namespace of a schema containing user-defined types and declarations
final String targetNamespace = "http://www.example.org/schema/";

// Register the schema with the XFactory
factory.registerSchema(schemaSource);

// Create a sequence type for exactly one document node with an element of type "employeeRecord" from the schema:
// "document-node(element(*, ns:employeeRecord))"
XSequenceType employeeRecordDocumentType = stFactory.documentNodeWithElement(
        null,
        new QName(targetNamespace, "employeeRecord"),
        false,
        XSequenceType.OccurrenceIndicator.ONE);

// Create a sequence type for an optional attribute matching the attribute declaration "type" in the schema:
// "schema-attribute(ns:type)?"
XSequenceType optionalEmployeeType = stFactory.schemaAttribute(
        new QName(targetNamespace, "type"),
        XSequenceType.OccurrenceIndicator.ZERO_OR_ONE);

// Create a sequence type for one or more atomic values of type "telephoneNumber" from the schema:
// "ns:telephoneNumber+"
XSequenceType telephoneNumbersType = stFactory.atomic(
        new QName(targetNamespace, "telephoneNumber"),
        XSequenceType.OccurrenceIndicator.ONE_OR_MORE);
```

Contents of sample schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:ns="http://www.example.org/schema/" targetNamespace="http://www.example.org/schema/">
    <complexType name="employeeRecord">
        <sequence>
            <element name="name" type="string"></element>
            <element name="id" type="int"></element>
            <element name="telephone" type="ns:telephoneNumber"></element>
        </sequence>
        <attribute ref="ns:type"></attribute>
    </complexType>

    <simpleType name="telephoneNumber">
        <restriction base="string">
            <pattern value="\d{3}-\d{3}-\d{4}"></pattern>
        </restriction>
    </simpleType>

    <element name="employee" type="ns:employeeRecord"></element>

    <simpleType name="employeeType">
        <restriction base="string">
            <enumeration value="full-time"></enumeration>
```

```
            <enumeration value="part-time"></enumeration>
            <enumeration value="seasonal"></enumeration>
        </restriction>
    </simpleType>

    <attribute name="type" type="ns:employeeType"></attribute>
</schema>
```

Working with collations

XSLT stylesheets and expressions in XQuery and XPath can refer to collations using collation URIs. A collation is a set of culture-specific rules that define how text should be sorted and which differences between two pieces of text are considered significant and which insignificant.

Before you begin

This article assumes some basic familiarity with the java.util.Locale and java.text.Collator classes.

About this task

The processor does not interpret the collation URI in any way -- it treats a collation URI merely as a sort of name for the instance of the Java Collator class that is associated with that URI. The XML API provides mechanisms for specifying what will be the default collation URI at preparation-time and for associating an instance of the Java Collator class with a collation URI at execution-time.

All collation URIs specified through the XML API must be absolute URI references. In an XSLT stylesheet or an XQuery or XPath expression, any relative URI reference that is used in a context where a collation URI is required will be resolved against the base URI from the static context for that expression -- that will ensure that even relative URI references in the stylesheet or expression can be matched with the absolute URI references specified through the XML API.

Limitations:

- If a collation URI is bound with an instance of the Java Collator class that is not an instance of java.text.RuleBasedCollator, certain operations will not be permitted with that collation URI. In particular, the fn:starts-with, fn:ends-with, fn:contains, fn:substring-before and fn:substring-after functions are not supported with that collation URI.
- All instances of Collator that are currently included with the Java runtime environment are also instances of java.text.RuleBasedCollator, so this is for most purposes only a theoretical limitation. However, it is something to be aware of if an application defines its own instances of the Java Collator class or defines subclasses the Collator class that are not also instances of java.text.RuleBasedCollator.

Procedure

- Declare the default collation URI.

  You can specify what collation URI you want to use as the default for string comparison operations by using the method setDefaultCollation method on the XStaticContext interface. The default collation URI from the XStaticContext interface will be used as the collation URI in string comparison operations that do not explicitly specify a collation URI.

  An XQuery expression can override the default collation URI specified on the XStaticContext interface with the declare default collation declaration. Similarly, an XSLT stylesheet can override the default collation URI with the [xsl:]default-collation attribute. XPath does not provide a means of overriding the default collation URI. However, any XPath or XQuery expression or XSLT stylesheet that performs string comparison operations can specify an explicit collation URI to override the default collation URI.

  If you do not explicitly specify a default collation on any instance of the XStaticContext interface you supply when you prepare your XSLT stylesheet or your XQuery or XPath expression, the default collation URI for the stylesheet or expression will be the Unicode code-point collation URI: http://www.w3.org/2005/xpath-functions/collation/codepoint/.

You can use the Unicode code-point collation in situations where characters must be identical Unicode characters to be considered to be equal. The lexicographical ordering defined by this collation is determined by the Unicode code points of the characters -- that is, by their positions on the Unicode code charts. As such, using the Unicode code-point collation will yield much better performance than collations that perform string comparisons in a culture-specific manner, but its unlikely to give very satisfactory results for sorting operations.

The following is a simple example showing how to specify the default collation URI on an instance of the XStaticContext interface.

```
// Setting of default collation URI is not changed - default remains
// the Unicode code point collation URI
XFactory factory = XFactory.newInstance();
XPathExecutable maxPath1 = factory.prepareXPath("max($var)");

// A new default collation URI is specified in the static context
// That URI is used in any string comparison for which no other
// explicit collation URI is specified
XStaticContext sc = factory.newStaticContext();
sc.setDefaultCollation("http://example.org/my-collation");
XPathExecutable maxPath2 = factory.prepareXPath("max($var)", sc);
```

- Bind a collation URI.

  The XML API provides two methods for binding collation URI with an instance of the Java Collator class for an execution. The bindCollation methods on the XDynamicContext method have two arguments: the first argument is a collation URI; the second is either instance of the java.text.Collator class or an instance of the java.util.Locale class. If an instance of the locale class is specified, the processor will use the instance of the Collator class that is appropriate for that locale.

  XSLT, XPath and XQuery define the concept of “Statically Known Collations”. If a reference to a collation URI appears in an XSLT stylesheet or an XPath or XQuery expression, and the collation URI is not one of the Statically Known Collations, a static error is supposed to be reported in some circumstances. However, the processor treats all collation URIs as if they were in the set of Statically Known Collations. This is due to the fact that instances of the Java Collator class are not actually associated with collation URIs until execution time, so it is not possible for the processor to determine statically which collation URIs are not known. Instead, the processor will report a dynamic error if a collation URI that is not bound to an instance of the Collator class is used in a stylesheet or expression.

  You cannot bind the Unicode code-point collation URI to any instance of the Java Collator class. It is always implicitly bound with the Unicode code-point collation.

  The following example demonstrates how you can bind a collation URI with a specific instance of the Java Collator class on an instance of the XDynamicContext interface.

```
XFactory factory = XFactory.newInstance();
XStaticContext sc = factory.newStaticContext();
// Set up a default collation URI
sc.setDefaultCollation("http://example.org/my-collation");

// Prepare an XPath expression that computes fn:max() using the
// collator associated with the default collation URI and again using
// the Unicode code point collation
String expr =
    "max($var)," +
    "max($var,'http://www.w3.org/2005/xpath-functions/collation/codepoint')";
XPathExecutable maxPath =
    factory.prepareXPath(expr, sc);
XDynamicContext dc = factory.newDynamicContext();
// Set the value of the variable $var
dc.bind(new QName("var"),
        new String[] {"encyclopaedia",
                      // U+00E6 is lower case latin ae ligature
                      "encyclop\u00E6dia",
                      "encyclopedia"});
// Set up a Collator for English that does not distinguish between
// capitals, lower-case letters and certain character variants
Collator english =
    (Collator) Collator.getInstance(Locale.ENGLISH).clone();
english.setStrength(Collator.SECONDARY);
// Evaluate the expression with that English collator associated with
// the default collation URI
dc.bindCollation("http://example.org/my-collation", english);
XSequenceCursor maxValues = maxPath.execute(dc);
// Print maximum values - expected results are
// encyclopedia for English collation and
// encyclop\u00E6dia for Unicode code point collation
if (maxValues != null) {
```

```
    do {
        System.out.println(maxValues.getStringValue());
    } while (maxValues.toNext());
}
```

Executing using the command-line tools

You can use the ExecuteXSLT tool to execute a stylesheet, use the ExecuteXPath tool to execute an XPath expression, and use the ExecuteXQuery tool to execute an XQuery expression.

Procedure

- **Execute XSLT**

  Location

  The product includes the following script that sets up the environment and invokes the tool.

  Syntax

  Parameters

  -outputfile *file*

  Specifies that output should go to the specified file

  By default, the output is sent to standard out.

  -baseURI *URI*

  Specifies the base URI of the containing element

  -useCompiler

  Specifies compiler modes

  If this parameter is not specified, the default behavior is to use interpreted mode.

  -bindVar name=*varName* **value=***varValue*

  Bind an atomic value to a variable (XPath, XQuery) or a param (XSLT)

  The value must be valid for the type that was specified in the static context (XPath), in the query (XQuery), or in the stylesheet (XSLT) for the same name.

  varName

  Name of the variable (expressed localPart,namespaceURI)

  Notes:

  - If the variable is in no namespace, the namespace URI should be omitted.
  - If the value of any option contains a blank space, enclose it in quotation marks.
  - Localpart is a required value.

  varValue

  Variable value

  The -bindVar option can be used multiple times.

  For example:

```
-bindVar name=la,"http://www.ibm.com/Los Angeles" value="some value"
```

  -baseOutputURI *URI*

  Specifies the base URI to use when resolving result documents

  The default is either the base URI for the main result document or the current working directory.

  This option only applies when working with XSLT.

-XSLTinitMode *mode*
Specifies a mode to use as the initial mode in an XSLT transformation (expressed localPart,namespaceURI)

This option only applies when working with XSLT.

For example:

```
-XSLTinitMode la,"http://www.ibm.com/Los Angeles"
```

-XSLTinitTemplate *template*
Specifies a named template to use as the initial template in an XSLT transformation (expressed localPart,namespaceURI)

If a named template is not set, the initial template is determined by the initial mode, context node, and template matching rules.

This option only applies when working with XSLT.

For example:

```
-XSLTinitTemplate la,"http://www.ibm.com/Los Angeles"
```

-v Prints the version of the tool

-h Prints the usage statement

-input *file*
Specifies the full path to a file containing an XML artifact against which the stylesheet will be executed

stylesheet
Specifies the full path to a file containing the XSL stylesheet

The following is a basic example of executing a stylesheet using the ExecuteXSLT tool:

- **Execute an XPath expression**

  Location
  The product includes the following script that sets up the environment and invokes the tool.

  Syntax

  Parameters

  -outputfile *file*
  Specifies that output should go to the specified file

  By default, the output is sent to standard out.

  -cpm *mode*
  Specifies an alternate XPath compatibility mode

  Valid values are Latest, 1.0, and 2.0.

  The default is 2.0.

  -ns *prefix*=*URI*
  Specifies a namespace for use during static processing

  If the value of any option contains a blank space, enclose it in quotation marks.

  This option can be used multiple times.

  -schema *URI*
  Specifies any schema document that is used to populate the in-scope schema definitions

  This option can be used multiple times.

-var name=*varName* **type=***varType*

Adds a variable binding to the static context for a single item

This simply declares the variable, and a value also must be bound to the XDynamicContext.

varName

Indicates the name of the variable (expressed localPart,namespaceURI)

If the variable is in no namespace, the namespace URI should be omitted.

Localpart is a required value.

varType

Indicates the type of the variable (expressed localPart[, namespaceURI])

If the value of any option contains a blank space, enclose it in quotation marks.

This option can be used multiple times.

For example:

```
-var name=la,"http://www.ibm.com/Los Angeles" type=boolean,http://www.w3.org/2001/XMLSchema
```

-baseURI *URI*

Specifies the base URI of the containing element

-dnet *URI*

Specifies a default namespace URI for element and type names

The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.

-useCompiler

Specifies compiler modes

If this parameter is not specified, the default behavior is to use interpreted mode.

-bindVar name=*varName* **value=***varValue*

Bind an atomic value to a variable (XPath, XQuery) or a param (XSLT)

The value must be valid for the type that was specified in the static context (XPath), in the query (XQuery), or in the stylesheet (XSLT) for the same name.

varName

Name of the variable (expressed localPart,namespaceURI)

Notes:

- If the variable is in no namespace, the namespace URI should be omitted.
- If the value of any option contains a blank space, enclose it in quotation marks.
- Localpart is a required value.

varValue

Variable value

The -bindVar option can be used multiple times.

For example:

```
-bindVar name=la,"http://www.ibm.com/Los Angeles" value="some value"
```

-v Prints the version of the tool

-h Prints the usage statement

-input *file*
Specifies the full path to a file containing an XML artifact against which the XPath expression will be executed

xpathfile
Specifies the full path to a file containing the XPath expression

The following is a basic example of executing an XPath expression using the ExecuteXPath tool:

- **Execute an XQuery expression**

Location
The product includes the following script that sets up the environment and invokes the tool.

Syntax

Parameters

-outputfile *file*
Specifies that output should go to the specified file

By default, the output is sent to standard out.

-baseURI *URI*
Specifies the base URI of the containing element

-dnet *URI*
Specifies a default namespace URI for element and type names

The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.

-useCompiler
Specifies compiler modes

If this parameter is not specified, the default behavior is to use interpreted mode.

-bindVar name=*varName* **value=***varValue*
Bind an atomic value to a variable (XPath, XQuery) or a param (XSLT)

The value must be valid for the type that was specified in the static context (XPath), in the query (XQuery), or in the stylesheet (XSLT) for the same name.

varName
Name of the variable (expressed localPart,namespaceURI)

Notes:

- If the variable is in no namespace, the namespace URI should be omitted.
- If the value of any option contains a blank space, enclose it in quotation marks.
- Localpart is a required value.

varValue
Variable value

The -bindVar option can be used multiple times.

For example:

```
-bindVar name=la,"http://www.ibm.com/Los Angeles" value="some value"
```

-v Prints the version of the tool

-h Prints the usage statement

-input *file*
Specifies the full path to a file containing an XML artifact against which the XQuery expression will be executed

xqueryfile
Full path to a file containing the XQuery expression

The following is a basic example of executing an XQuery expression using the ExecuteXQuery tool:

Using a message handler and managing exceptions

You can use this information to help you to use a message handler and mange exceptions.

Procedure

- Use a message handler.
- Manage exceptions.

Using a message handler

The default behavior for prepare-time and execution-time processing errors and other messages is to print the messages to System.err and for nonrecoverable errors to raise an XProcessException as well. If an error occurs at prepare time, the processor attempts to continue preparation and signal all errors before generating an XProcessException; but no executable is produced. At run time, execution stops at the first occurrence of an error situation.

Procedure

Change the handling of errors by registering an implementation of XMessageHandler to the XStaticContext (for prepare-time errors and other messages) or the XDynamicContext (for execution-time errors and other messages).
The XMessageHandler interface consists of a single report method that has the following parameters:

level One of the following enumerators defined in the XMessageHandler interface:

INFO Indicates that the error is simply informational and will not affect the result

This is also used for the XSLT message instruction when the terminate attribute evaluates to "no."

WARNING
Indicates a warning

The processor recovers from a warning situation, but the result might not be what was expected.

ERROR
Indicates a recoverable error

The processor might recover from this error for the purpose of signaling additional errors, but no result is produced.

FATAL_ERROR
Indicates a nonrecoverable error

The processor cannot recover from this error. It is also used for the XSLT message instruction when the terminate attribute evaluates to "yes."

TRACE
Indicates that the message was generated by a call to the XPath fn:trace function

message
Error message

location
Source location as an XSourceLocation if available

In general, the source location is not available for execution-time errors.

cause Original exception that caused the error if available

If the input document is invalid, for example, the XML parser generates an exception that is passed to the report method through this parameter.

errorItems
Items that were specified for the error-object parameter to the XPath fn:error function

The implementation of XMessageHandler can present the errors and other messages as desired, such as writing to a log file rather than sending the messages to System.err. It may also be more strict and stop compilation or execution after any error including recoverable errors by generating an exception. Because the report method has no throws clause, the exception must be unchecked. The implementation might also choose to ignore informational and warning messages. In short, registering an XMessageHandler allows the application to configure message handling to suit its purposes.
Note that in the case of a nonrecoverable error, if the registered message handler does not generate an exception, an XProcessException is raised by the processor.

Managing exceptions

Various exceptions might occur. Unless otherwise indicated, all exceptions extend RuntimeException and thus are unchecked.

Procedure

- Manage **XProcessException**.

  This exception occurs when the processor finds a nonrecoverable error when preparing or executing an expression, query, or stylesheet as described in the specifications for each language.

  If there are multiple errors at prepare time, the processor attempts to report all of the errors and only generates an XProcessException at the end of preparation or if it reaches a point where it cannot continue. At execution time, however, the first error results in an XProcessException and the end of execution.

  In general, XProcessExceptions should not occur if the expression, query, or stylesheet is syntactically and semantically valid and also valid for the types of input documents that it is meant to process.

  An XProcessException also occurs for an XSLT message instruction where the terminate attribute evaluates to "yes."

  This is the default behavior for handling processing errors and other messages. Applications can register an implementation of the XMessageHandler interface at prepare time or execution time to modify the default behavior.
- Manage an **XViewException**.

  This exception is raised for incorrect use of the XML API itself such as calling one of the XNodeView methods on an item that is atomic.

  See the API documentation for more information about when this exception can occur.
- Manage a **NullPointerException**.

  This exception is raised when a null value is passed to an API method where null is not allowed, such as passing a null prefix or namespace to the XStaticContext declareNamespace method.

  See the XML API documentation for more information about when this exception can occur.
- Manage an **IllegalArgumentException**.

  This exception is raised when an invalid value is passed to an API method, such as calling the XStaticContext setXPathCompatibilityMode method with a value other than one of the predefined settings.

  See the XML API documentation for more information about when this exception can occur.

- Manage **exceptions generated by XFactory.newInstance**.

  The newInstance method on the XFactory class loads the XFactory implementation class and creates a new instance; therefore, any of the following checked exceptions might occur:

  – ClassNotFoundException
  – IllegalAccessException
  – InstantiationException

  These exceptions would likely result if there is a classpath problem or a Java security issue.

Chapter 40. Deploying client applications

Deploying a client application depends on installing appropriate supporting files on the client machine, usually some configuring actions, and adding the program files for the client application. When the client application has been deployed, the application can run.

About this task

The steps required to deploy and run a client application depend on the type of client and the programming model used.

You can install an application client JAR file using the administrative console, wsadmin AdminApp install, or update commands. Install the client module only on a Version 8.0 deployment target (such as server, cluster, and so on).

Complete one or more of the following tasks:

Procedure

- Deploy the client application
- Run an ActiveX client application
- Deploy and run a Java EE client application
- Run the IBM Thin Client for Enterprise JavaBeans

Deploying applet client code

Applet clients are capable of communicating over the HTTP protocol and the RMI-IIOP protocol.

Before you begin

Applet clients have the following setup requirements:
- These clients are available on the Windows platforms. Check the prerequisites page for information on platform support and product prerequisites.
- The browser installation precedes the client code installation.

About this task

Unlike typical applets that are on web servers or WebSphere Application Servers and can only communicate over the HTTP protocol, applet clients can communicate over the HTTP protocol and the RMI-IIOP protocol. This additional capability gives the applet direct access to enterprise beans.

The applet container is the web browser and the Java plug-in combination. You must first install the Application Client for WebSphere Application Server so that the browser recognizes the IBM product Java plug-in.

Procedure

1. Install the Application Client for WebSphere Application Server.
2. Configure required Java runtime parameters.
   a. Click **Start** > **Control panel** .
   b. Select the IBM Control Panel for Java
   c. On the Advanced tab, enter the following parameter values in the Java Runtime Parameters field.

```
-Xmx512M
-Djava.security.policy=<app_client_root>\properties\client.policy
-Dwas.install.root=<app_client_root>
-Djava.ext.dirs=<app_client_root>\java\jre\lib\ext;
<app_client_root>\lib;
<app_client_root>\plugins;
<app_client_root>\lib\ext;
<app_client_root>\installedConnectors\
-Djava.class.path=<app_client_root>\properties
-Dcom.ibm.CORBA.ConfigURL=file:<app_client_root>\properties\sas.client.props
-Dcom.ibm.SSL.ConfigURL=file:<app_client_root>\properties\ssl.client.props
```

Note: These parameter entries are automatically placed into the WebSphere Application Server control panel for the Java plug-in user who installed the WebSphere Application Server Application Client provided you are using a Java SE Development Kit (JDK) prior to JDK 1.5. If the applet is being run by a user other than the person who installed the client, then that user must enter the parameter entries.

For JDK 1.5 and later, this automatic parameter feature is removed.

- The Java Runtime Parameters field is similar to the command prompt when using command line options. Therefore, you can enter most options available from the command prompt (for example, -cp, classpath, and others) in this field as well.

3. Configure use of secure sockets layer (SSL) for secure access to resources. By default, the applet client is configured to have security enabled. If you have administrative security turned on at the server from which you are accessing resources, then you can use SSL when needed. If you decide that the security requirements for applet client applications differ from other types of client applications, then you can create special copies of client property files for applets to use.

Running an ActiveX client application

To run an ActiveX client application that is to use the ActiveX to Enterprise Java Beans (EJB) bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its `XJB.JAR` file and the Java run time. This initial configuration sets up the environment within which the ActiveX client application can run.

About this task

To perform the required configuration, complete one or more of the following tasks:

Procedure

1. Start an ActiveX application and configure service programs.
2. Start an ActiveX application and configuring non-service programs

Starting an ActiveX application and configuring service programs

To run an ActiveX service program such as Active Server Page (ASP) that is to use the ActiveX to the Enterprise Java Bean (EJB) bridge, some initial configuration (to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its `XJB.JAR` file and the Java run time) is necessary. This configuration sets up the environment within which the ActiveX service program can run.

Before you begin

The XJB.JClassFactory must find the Java run time dynamic link library (DLL) when initializing. In a service program such as Internet Information Server you cannot specify a path for its processes independently; you must set the process paths in the system PATH variable. This limitation means that you can only have a single Java virtual machine (JVM) version available on a machine using ASP.

About this task

To add the Java Runtime Environment (JRE) directories to your system path, complete one of the following task.

Procedure

On Windows XP systems, complete the following steps:
1. Open the Control Panel, then double-click the **System** icon.
2. Click the **Advanced** tab on the System Properties window.
3. Click **Environment Variables**.
4. Edit the Path variable in the System Variables window.
5. Add the following information to the beginning of the path that is displayed in the Variable Value field:

   `C:\WebSphere\AppClient\Java\jre\bin;C:\WebSphere\AppClient\Java\jre\bin\classic;`

   where `C:\WebSphere\AppClient` is the directory in which you installed the Java client in the WebSphere product.
6. Click **OK** in the Edit System Variable window to apply the changes.
7. Click **OK** in the Environment Variables window.
8. Click **OK** in the System Properties window.
9. Restart Windows XP.

What to do next

After you change the system PATH variable you must reboot the Internet Information Server machine so that Internet Information Server can see the change.

Starting an ActiveX application and configuring non-service programs

To run an ActiveX program initiated from an icon or command line (a non-service program) that is to use the ActiveX to the Enterprise Java Beans (EJB) bridge, you must perform some initial configuration to set appropriate environment variables and to enable the ActiveX to EJB bridge to find its `XJB.JAR` file and the Java run-time environment. This uses a batch file to set up the environment within which the ActiveX program can run.

About this task

To perform the required configuration, complete the following steps:

Procedure

1. Edit the `setupCmdLineXJB.bat` file to specify appropriate values for the environment variables required by the ActiveX to EJB bridge. For more information about these environment variables, see ActiveX to EJB bridge, environment and configuration. For more information about creating a JVM for an ActiveX program, see ActiveX to EJB bridge, initializing the Java virtual machine (JVM). After the ActiveX program has created an XJB.JClassFactory object and called the XJBInit() method, the JVM is initialized and ready for use.
2. Start the ActiveX client application by using one of the following methods:
   - Use the `launchClientXJB.bat` file to start the application. For example:

     ```
     launchClientXJB MyApplication.exe parm1 parm2
     ```

     or

     ```
     launchClientXJB MyApplication.vbp
     ```
   - Use the `setupCmdLineXJB.bat` file to create an environment in which to run the application, then start the application from within that environment.

setupCmdLineXJB.bat, launchClientXJB.bat and other ActiveX batch files

This topic provides reference information about the aids that client applications and client services can use to access the ActiveX to EJB bridge. These enable the ActiveX to Enterprise JavaBeans (EJB) bridge to find its XJB.JAR file and the Java run-time environment.

Location

The include file is located in the was_client_home\aspIncludes directory. You can include the file into your Active Server Pages (ASP) application with the following syntax in your ASP page:

```
<--  #include virtual ="/WSASPIncludes/setupASPXJB.inc" -->
```

This syntax assumes that you have created a virtual directory in Internet Information Server called WSASPIncludes that points to the was_client_home\aspIncludes directory.

Usage notes

The following batch files are provided for client applications to use the ActiveX to EJB bridge:

- setupCmdLineXJB.bat

  Sets the client environment variables.
- launchClientXJB.bat

  Calls the setupCmdLineXJB.bat file and launches the application you specify as its arguments; for example:

  ```
  launchClientXJB.bat myapp.exe parm1 parm2
  ```

  or

  ```
  launchClientXJB MyApplication.vbp
  ```
- Active Server Pages (ASP) include file

  An include file is provided for ASP users to automatically set the following page-level (local) environment variables:
  - com_ibm_websphere_javahome. Path to the Java run-time directory installed with the WebSphere advanced server client.
  - com_ibm_websphere_washome. Path to the WebSphere advanced server client directory.
  - com_ibm_websphere_namingfactory. Sets the Java java.naming.factory.initial system property.
  - com_ibm_websphere_computername. (Optional) Name of the computer where the WebSphere Advanced Server Client is installed. If you intend to talk to a single specific computer, you are recommended to change this value to become the server name that you intend to access.
- System settings

  To enable the ActiveX to EJB bridge to access the Java run-time dynamic link library (DLL), the following directories must exist in the system PATH environment variable:

  ```
  was_client_home\java\jre\bin;was_client_home\java\jre\bin\classic
  ```

  Where was_client_home is the name of the directory where you installed the WebSphere Application Server client (for example, C:\WebSphere\AppClient).

  Note: This technique enables only one Java run time to activate on a machine, therefore all client services on that machine must use the same Java run time. Client applications do not have this limitation because they each have their own private, non-system scope.

Deploying and running a Java EE client application

You can use the **launchClient** command to run a Java Enterprise Edition (EE) client application in an Application Client installation or in a WebSphere Application Server node. Alternatively, you can use Java Web Start on a remote client machine to download and run a Java EE client application, including Thin client application, with a single click from a web browser on that machine.

Procedure

1. Deploy and run a Java EE client application for use with the **launchClient** command.

   After deploying a Java EE client application onto a machine with an Application Client installation or in a WebSphere Application Server node, you can start the application by using the **launchClient** command on that machine.

   a. Deploy the Java EE client application

   b. Start the Java EE client application

2. Deploy and run a Java EE client application by using Java Web Start.

   a. Prepare the Java EE client application ready to be deployed by remote action.

   b. Use Java Web Start on a remote client machine to download and run the Java EE client application.

Deploying a Java EE client application

Deploying a Java EE client application onto the client machines where it is to run includes distributing the EAR file for the client application and configuring resource references for use by the client application.

Before you begin

To run a deployed Java EE client application, the application needs access to a Application Client installation or a WebSphere Application Server installation.

For information about installing the Application Client on a client machine, refer to the Installing Application Client for WebSphere Application Server topic.

Attention: Application Client for WebSphere Application Server ships only with the 32-bit WebSphere Application Server.

About this task

Use this topic only if you later want to use the want to **launchClient** command to run the Java client application on an Application Client installation or in a WebSphere Application Server node.

If you want to download and run a Java EE client application remotely, you can use the Java Web Start to deploy the application onto the remote client machine with a single click from a Web browser on the client machine. For information about using Java Web Start to deploy Java EE client applications, see “Downloading and running a Java EE client application by using Java Web Start”.

Procedure

1. Distribute the EAR file.

   The client machines configured to run a client application must have access to the EAR file.

   - If all the machines in your environment share the same image and platform, run the Application Client Resource Configuration Tool (ACRCT) on one machine to configure the external resources, then distribute the configured EAR file to the other machines.
   - If your environment is set up with a variety of client installations and platforms, run the ACRCT for each unique configuration.

- You can either distribute an EAR file to the correct client machines, or make it available on a network drive.
- Distributing EAR files is the responsibility of the system and network administrator.

2. Configure the resources for the application client. This generally involves using the Application Client Resource Configuration Tool (ACRCT) to configure references for the resources that the application is to use, including resource adapters, resource providers, data sources, and Java Message Service resources. These configurations are stored in the client JAR file within the application EAR file. The client runtime uses these configurations to resolve and create an instance of the resources for the client application.

   For some types of resources, other actions are needed; for example, to install a resource adapter and define environment variable needed to start the client application. More information about the actions for different types of resources is given in other configuring resources topics.

   If you plan to deploy the client application on z/OS, run the ACRCT on Windows. You can also run the ACRCT for distributed platforms locally.

   If the client application defines the local resources, but the resources are installed in a different location, run the ACRCT (**clientConfig** command) on the local machine to change the configuration in the EAR file. For example, the EAR file can contain a DB2 resource, configured as `C:\DB2`. If, however, you installed DB2 in the `D:\Program Files\DB2` directory, use the ACRCT to create a local version of the EAR file.
3. Use the WebSphere Administrative console to install the client application on z/OS.

What to do next

After deploying the Java EE client application, use the **launchClient** command to run the client application.

Starting the Application Client Resource Configuration Tool and opening an EAR file

You can perform many tasks by starting the Application Client Resource Configuration Tool (ACRCT). Many of these tasks also involve then opening an EAR file.

Before you begin

Attention: This task only applies to Java Platform, Enterprise Edition (Java EE) application clients.

About this task

Use these steps to start the Application Client Resource Configuration Tool. When you start the tool, one of the most common tasks that you perform is opening and modifying the components of EAR files.

Procedure

1. Open a command prompt and change to the *app_server_root*`\bin` directory.
2. Run the `clientConfig.bat` file.
3. Open an EAR file within the Application Client Resource Configuration Tool (ACRCT):
   a. Click **File** > **Open**.
   b. Select the file then click **Open**.
4. Save your changes to the file and close the tool:
   a. Click **File** > **Save**.
   b. Click **File** > **Exit**.

Deploying a resource adapter for a Java EE client application

A Java EE client application can use a resource adapter to connect to an enterprise information system (EIS). To use a resource adapter, you need to install it, configure it, and configure related resources.

About this task

The resource adapter support provided for Java EE client applications is a subset of the support provided for application servers. A client resource adapter is used in a non-managed environment and must conform to the J2EE Connector Architecture Specification Version 1.5 or higher. Only outbound connections to the EIS are supported through the ManagedConnectionFactory interfaces. The inbound messaging support (from the EIS), life cycle management, and work management aspects of the specification are not supported on the client.

When running Java EE application clients, the **launchClient** script specifies a system property called **com.ibm.ws.client.installedConnector**, which is set to the same value as the *CLIENT_CONNECTOR_INSTALL_ROOT* variable. This is the default location for installed resource adapters and can be overridden for each **launchClient** call by specifying the **-CCD** parameter. When the client container is activated, all resource adapter subdirectories under the specified default location for the resource adapters directory are added to the classpath. This action allows the client application to use the resource adapters without using the ACRCT to specify any of the client resources.

Procedure

1. Install the resource adapter archive (RAR) file

   For a client application to use a resource adapter, the RAR file must be installed in the directory specified by the environment variable, CLIENT_CONNECTOR_INSTALL_ROOT, defined when the **setupCmdLine** script runs. The **launchClient** tool, Application Client Resource Configuration Tool (ACRCT) and clientRAR tool all use this variable to find the default location of all installed resource adapters.

   To install a RAR file for a client application, use the clientRAR tool.
2. Configure the resource adapter and its resources for the client application Use the Application Client Resource Configuration Tool (ACRCT) to define the resource adapter, connection factories, and administered objects in the EAR file for the client application. The client application uses this configuration to resolve and create an instance of the resource adapter and the other resources.
   a. Configure the resource adapter
   b. Configure a connection factory
   c. Configure administered objects

clientRAR tool:

This topic describes the command line syntax for the client resource adapter installation tool.

If this tool is used to add or delete resource adapters on the server, then only the client can use the resource adapter. If the resource adapter is installed on the server using the wsadmin tool or the administrative console, then do not use the clientRAR tool remove it. Only resource adapters that are installed using the clientRAR tool should be removed using the clientRAR tool.

The command line invocation syntax for the `clientRAR` tool follows:

```
clientRAR [-help | -?] [-CRDcom.ibm.ws.client.installedConnectors=<dir>] <task> <archive>

where
-help, -?
Print the usage information.
-CRDcom.ibm.ws.client.installedConnectors
The directory where resource adapters are installed.
This will override the system property of the same name
(com.ibm.ws.client.installedConnectors).


<task>
The task to perform: add - install, delete - uninstall.
```

```
<archive>
if task=add then this is the fully qualified name of the resource adapter archive file.
If task=delete then this is the filename of the resource adapter archive to be uninstalled.
```

The following examples demonstrate correct syntax.

Configuring resource adapters for the client:

Use the Application Client Resource Configuration Tool (ACRCT) to configure resource adapters for the client.

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure new resource adapters. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new resource adapters from the tree.
4. Expand the JAR file to view its contents.
5. Right-click the Resource Adapters folder, and click **New**.
6. Configure the resource adapter settings in the resulting property dialog.
7. Click **OK**.
8. Click **File > Save** on the menu bar to save your changes.

Resource adapters for the client:

A resource adapter is a system-level software driver that a Java application uses to connect to an enterprise information system (EIS). A resource adapter plugs into an application client and provides connectivity between the EIS and the enterprise application.

Important: This topic is not relevant to the WebSphere MQ resource adapter. WebSphere MQ classes are picked up automatically by the client container (for both stand alone and with a WebSphere Application Server installation).

The resource adapter support for the Java EE client applications is a subset of the support for the server. For any resource adapter installed using the clientRAR tool, the client resource adapter is used in a non-managed environment and must conform to the Java EE Connector Architecture Specification Version 1.5 or higher. Only outbound connections to the EIS are supported through the ManagedConnectionFactory interfaces. The inbound messaging support (from the EIS), life cycle management, and work management aspects of the specification are not supported on the client.

For a client application to use a resource adapter, it must be installed in the directory specified by the environment variable, CLIENT_CONNECTOR_INSTALL_ROOT, defined when the setupCmdLine script runs. The launchClient tool, Application Client Resource Configuration Tool (ACRCT) and clientRAR tool all use this variable to find the default location of all installed resource adapters. To install a resource adapter in the client, use the clientRAR tool. Once the resource adapter is installed, it must be configured using the ACRCT. The client configuration tool adds the resource adapter configuration to the EAR file. Then, connection factories and administered objects are defined.

When running Java EE application clients, the `launchClient` script specifies a system property called `com.ibm.ws.client.installedConnector`, which is set to the same value as the `CLIENT_CONNECTOR_INSTALL_ROOT` variable. This is the default location for installed resource adapters and can be overridden for each `launchClient` call by specifying the `-CCD` parameter. When the client container is activated, all resource adapter subdirectories under the specified default location for the resource adapters directory are added to the classpath. This action allows the client application to use the resource adapters without using the ACRCT to specify any of the client resources.

Using resource adapters is a new mechanism for easily extending client applications.

Resource adapter settings:

Use this panel to view or change the configuration properties of the resource adapter. These configuration properties control how resource adapters are created.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Adapter**. Right-click **Resource Adapter** and click **New**. The following fields appear on the **General** tab.

Name:

The name by which this Resource Adapter is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the Resource Adapters across the product administrative domain.

Information	Value
Data type	String

Description:

A description of this resource adapter for administrative purposes within IBM WebSphere Application Server.

Information	Value
Data type	String

Class Path:

Any additional class path. The path to the resource adapter directory is automatically added.

Information	Value
Data type	String
Default	The path to your Resource Adapter directory.

Native Path:

The native path where the Resource Adapter is located. Enter any additional native class path here.

Information	Value
Data type	String

Resource Adapter Name:

A mandatory field that points to an installed resource adapter subdirectory. The entry does not represent the full directory name for the resource adapter. The full directory name is the installed resource adapter path, plus the resource adapter name.

Information	Value
Data type	String

Installed Resource Adapter Path:

The directory where resource adapters are installed. If you do not complete this field, then the default takes effect.

If you specify the value, `${CONNECTOR_INSTALL_ROOT}`, then this value replaces the value of the `CLIENT_CONNECTOR_INSTALL_ROOT` variable on the machine on which the client application runs. This action allows the application to run easily on different machines, where the client installation might be in different locations.

Information	Value
Data type	String
Default	${CONNECTOR_INSTALL_ROOT}

Configuring new connection factories for resource adapters for the client:

Use the Application Client Resource Configuration Tool (ACRCT) to configure new connection factories for resource adapters for the client.

About this task

Complete this task to configure new connection factories for resource adapters.

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure new connection factories. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new connection factories from the tree.
4. Expand the JAR file to view its contents.
5. Click the Resource Adapters folder.
6. Expand the resource adapter for which you want to create connection factories.
7. Right-click the Connection Factories folder and click **New**.
8. Configure the connection factory properties in the resulting property dialog.
9. Click **OK**.
10. Click **File > Save** on the menu bar to save your changes.

Resource adapter connection factory settings:

Use this panel to view or change the configuration properties of the selected resource adapter connection factory.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Adapters**. Right-click the **Connection Factories** folder, and click **New**. The following fields appear on the **General** tab.

Name:

The name by which this connection factory is known for administrative purposes within WebSphere Application Server. The name must be unique within the resource adapter connection factories across the product administrative domain.

Information	Value
Data type	String

Description:

An optional description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The JNDI name that is used to match this resource adapter connection factory definition to the deployment descriptor. This entry should be a `resource-ref` name.

Information	Value
Data type	String

User Name:

The **User Name** used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly when getting a connection. If this field is used, then the Properties field `UserName` is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User Name** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly when getting a connection.

Information	Value
Data type	String

Password:

Specifies an encrypted password. If you complete this field, then the **Password** field in the Properties box is ignored.

If you specify a value for the **UserName** property, you must also specify a value for the **Password** property.

Information	Value
Data type	String

Re-Enter Password:

Confirms the password.

Type:

A drop-down list of all the `connectionFactoryInterfaces` as defined for the factories in the Resource Adapter Archive.

For each **Type**, there is a set of properties specified in the Properties box. This set of properties is constructed by retrieving the properties from each connection definition object. For any existing connection factories that are displayed for updating, this list of properties is overlaid with the properties specified for the objects. When the **Type** field is changed, the properties also change to reflect the correct properties for that type.

Information	Value
Data type	String

Configuring administered objects for resource adapters for the client:

This section helps you configure new administered objects for the client.

Before you begin

Before you configure new administered objects, you must complete the following prerequisites:

1. Install the Resource Adapter Archive file (RAR) using the clientRAR tool.
2. Configure the resource adapter for the `.ear` file, using the Application Client Resource Configuration Tool (ACRCT) tool.

About this task

Complete this task to configure new administered objects for installed resource adapters.

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure new administered objects. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new administered objects from the tree.
4. Expand the JAR file to view its contents.
5. Click the **Resource Adapters** folder.
6. Expand the resource adapter for which you want to create administered objects.
7. Right-click the **Administered Objects** folder and click **New**.
8. Configure the administered object properties in the resulting property dialog.
9. Click **OK**.
10. Click **File > Save** on the menu bar to save your changes.

Administered objects settings:

Use this panel to view or change the configuration properties of the selected administered objects.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Adapters** > *resource_adapter_instance*. Right-click **Administered Objects** and click **New**. The following fields appear on the **General** tab.

The settings for administered objects are handled similarly to connection factories. When updating administered objects, use the same panels that you used to create administered objects.

Name:

The name by which this administered object is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the resource adapter administered objects across the product administrative domain.

Information	Value
Data type	String

Description:

An optional description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

This entry is a `resource-env-ref` name, a `message-destination-ref` name (if the `message-destination-ref` has no link), or a `message-destination` link.

Information	Value
Data type	String

Type:

A drop-down list of all the administered object class-interface pairs as defined for the admin objects in the Resource Adapter Archive (RAR) file.

For each **Type**, there is a set of properties specified in the Properties box. This set of properties is constructed by retrieving the properties from each administered object definition. For any existing administered objects that are displayed for updating, this list of properties is overlaid with the properties specified for the objects. When the **Type** field is changed, the properties also change to reflect the correct properties for that type.

Information	Value
Data type	String

Enabling client use of data sources

If a Java EE client application accesses a database directly, you must provide the database drivers on the client machine, and configure the data source provider (JDBC provider) and data sources. Instead of accessing the database directly, it is recommended that your client application access the database through an enterprise bean.

About this task

WebSphere Application Server and the Application Client for WebSphere Application Server do not provide client database drivers to be used directly from a Java EE client application. You can contact your database vendor to get client database driver code and licenses.

Data sources configured on the server and looked up on the client do not participate in global transactions.

Instead of accessing the database directly, it is recommended that your client application access the database through an enterprise bean. This technique eliminates the need to have database drivers on the client machine, because the database access is handled by the enterprise bean running on WebSphere Application Server. It also enables the client application to take advantage of the pooling and additional database functions provided by the server.

For a current list of data source providers that are supported on WebSphere Application Server, see the WebSphere Application Server prerequisite website.

Procedure

1. For direct access from a client to the database, install the client database drivers on the client machine. For information about installing database drivers, see the documentation provided by your database vendor.
2. Configure a data source provider and a data source for the client application Use the Application Client Resource Configuration Tool (ACRCT) to define the data source provider and a data source in the EAR file for the client application. The client application uses this configuration to resolve and create an instance of the data source provider and data source.
   a. Configure a new data source provider. This provider describes the JDBC database implementation for your client application.
   b. Configuring a new data source This describes the client properties of the database your client application uses.

Configuring new data source providers (JDBC providers) for application clients:

You can create new data source providers, also known as JDBC providers, for your application client using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create new data source providers, also known as JDBC providers, for your application client. In a separate administrative task, install the Java code for the required data source provider on the client machine on which the application client resides.

About this task

Use this task to connect application clients to relational databases.

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file for which you want to configure the new data source provider. The EAR file contents display in a tree view.
2. Select the JAR file in which you want to configure the new data source provider from the tree.
3. Expand the JAR file to view its contents.
4. Click the Data Source Providers folder. Do one of the following:
   - Right-click the folder and click **New Provider**.
   - Click **Edit > New** on the menu bar.
5. Configure the data source provider properties in the resulting property dialog.
6. Click **OK** when you finish.
7. Click **File > Save** on the menu bar to save your changes.

Example

You can configure data source provider and data source settings.

- Configuring data source provider and data source settings

The following code examples illustrates how to use configure data source provider and data source settings:

```
<resources.jdbc:JDBCProvider xmi:id="JDBCProvider_1" name="jdbcProvider:name"
description="jdbcProvider:description" implementationClassName="jdbcProvider:
ImplementationClass">
<classpath>jdbcProvider:classPath</classpath>
<factories xmi:type="resources.jdbc:WAS40DataSource" xmi:id="WAS40DataSource_1"
name="jdbcFactory:name" jndiName="jdbcFactory:jndiName"
description="jdbcFactory:description" databaseName="jdbcFactory:databasename">
```

```
<propertySet xmi:id="J2EEResourcePropertySet_13">
<resourceProperties xmi:id="J2EEResourceProperty_13" name="jdbcFactory:customName"
value="jdbcFactory:customValue"/>
<resourceProperties xmi:id="J2EEResourceProperty_14" name="user"
value="jdbcFactory:user"/>
<resourceProperties xmi:id="J2EEResourceProperty_15" name="password"
value="{xor}NTs9PBk+PCswLSZlMT4yOg==/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_14">
<resourceProperties xmi:id="J2EEResourceProperty_16" name="jdbcProvider:customName"
value="jdbcProvider:customeValue"/>
</propertySet>
</resources.jdbc:JDBCProvider>
```

- Required fields:
  - Data Source Provider Properties page: name
  - Data Source Properties page: name, jndiName
- Special cases:
  - The user name and password fields have no equivalent XMI tags. You must specify these fields in the custom properties.
  - The password is encrypted when you use the Application Client Resource Configuration Tool (ACRCT). If you do not use the ACRCT the field cannot be encrypted.

Example: Configuring data source provider and data source settings:

You can configure data source provider and data source settings.

The purpose of this article is to help you to configure data source provider and data source settings.

- Required fields:
  - Data Source Provider Properties page: name
  - Data Source Properties page: name, jndiName
- Special cases:
  - The user name and password fields have no equivalent XMI tags. You must specify these fields in the custom properties.
  - The password is encrypted when you use the Application Client Resource Configuration Tool (ACRCT). If you do not use the ACRCT the field cannot be encrypted.
- Example:

```
<resources.jdbc:JDBCProvider xmi:id="JDBCProvider_1" name="jdbcProvider:name"
description="jdbcProvider:description" implementationClassName="jdbcProvider:
ImplementationClass">
<classpath>jdbcProvider:classPath</classpath>
<factories xmi:type="resources.jdbc:WAS40DataSource" xmi:id="WAS40DataSource_1"
name="jdbcFactory:name" jndiName="jdbcFactory:jndiName"
description="jdbcFactory:description" databaseName="jdbcFactory:databasename">
<propertySet xmi:id="J2EEResourcePropertySet_13">
<resourceProperties xmi:id="J2EEResourceProperty_13" name="jdbcFactory:customName"
value="jdbcFactory:customValue"/>
<resourceProperties xmi:id="J2EEResourceProperty_14" name="user"
value="jdbcFactory:user"/>
<resourceProperties xmi:id="J2EEResourceProperty_15" name="password"
value="{xor}NTs9PBk+PCswLSZlMT4yOg==/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_14">
<resourceProperties xmi:id="J2EEResourceProperty_16" name="jdbcProvider:customName"
value="jdbcProvider:customeValue"/>
</propertySet>
</resources.jdbc:JDBCProvider>
```

Data source provider settings for application clients:

Use this page to create a data source under a JDBC provider which provides the specific JDBC driver implementation class.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Data Source Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the display name for the data source.

For example you can set this field to *Test Data Source*.

Information	**Value**
Data type	String

Description:

Specifies a text description for the resource.

Information	**Value**
Data type	String

Class Path:

A list of paths or `.jar` file names which together form the location for the resource provider classes.

Implementation class:

Use this setting to perform database specific functions.

Information	**Value**
Data type	String
Default	Dependent on JDBC driver implementation class

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new data sources for application clients:

Learn how to create date sources for application clients.

About this task

During this task, you create new data sources for your application client.

Procedure

1. Click the data source provider for which you want to create a data source in the tree. Take one of the following actions as needed:
   - Configure a new data source provider.
   - Click an existing data source provider.
2. Expand the data source provider to view its Data Sources folder.
3. Click the data source folder. Take one of the following actions as needed:
   - Right click the data source folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the data source properties in the displayed fields.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Data source properties for application clients:

Use this page to create or modify the data sources.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Data Source Providers** > *Data source provider instance*. Right-click **Data Sources** and click **New**. The following fields are displayed on the **General** tab:

Name:

Specifies the display name of this data source.

Information	Value
Data type	String

Description:

Specifies a text description of the data source.

Information	Value
Data type	String

JNDI Name:

The application client run time uses this field to retrieve configuration information.

Database Name:

The name of the database to which you want to connect.

User:

Use the user ID with the Password property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the User ID property, then you must also specify a value for the Password property. The connection factory User ID and Password properties are used if the calling application does not provide a user ID and password explicitly.

Password:

Use the password with the User ID property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the Password property, then you must also specify a value for the User ID property.

Re-Enter Password:

Confirms the password.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring mail providers and sessions for application clients

You can edit the configurations of mail sessions and providers for your application clients using the Application Client Resource Configuration Tool (ACRCT).

About this task

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of mail sessions and providers for your application clients to use.

Procedure

1. Start the ACRCT.
2. Open an EAR file.
3. Locate the mail objects in the tree that is displayed for the EAR file. For example, if your file contains mail sessions, expand **Resources** > ***application*.jar** > **Mail Providers** > ***java_mail_provider_instance*** > **Mail Sessions**.

   In this example, ***java_mail_provider_instance*** is a particular mail provider.

Results

The mail session instances are located in the JavaMail Sessions folder.

Example

You can configure mail provider and mail session settings.

- Configuring mail provider and mail session settings for application clients

The following code examples illustrates how to configure mail provider and mail session settings for application clients:

```
<resources.mail:MailProvider xmi:id="builtin_mailprovider" name="Built-in Mail Provider" description="The built-in mail provider">
  <factories xmi:type="resources.mail:MailSession"
             xmi:id="MailSession_1207766754834" name="MailSession"
             jndiName="mail/session" description="Sample mail session" category="Sample"
             mailTransportHost="smtp.coldmail.com" mailTransportUser="transportUser"
             mailTransportPassword="{xor}Lz4sLChvLTs="
             mailFrom="smith@coldmail.com" mailStoreHost="imap.coldmail.com" mailStoreUser="storeUser"
             mailStorePassword="{xor}Lz4sLChvLTs="
             debug="true" strict="true"
             mailTransportProtocol="builtin_smtp" mailStoreProtocol="builtin_imap">
    <propertySet xmi:id="J2EEResourcePropertySet_1207766778585">
      <resourceProperties xmi:id="J2EEResourceProperty_1207766778585" name="key" type="java.lang.String" value="value" required="false"/>
```

```
    </propertySet>
  </factories>
  <protocolProviders xmi:id="builtin_smtp" protocol="smtp" classname="com.sun.mail.smtp.SMTPTransport" type="TRANSPORT"/>
  <protocolProviders xmi:id="builtin_pop3" protocol="pop3" classname="com.sun.mail.pop3.POP3Store" type="STORE"/>
  <protocolProviders xmi:id="builtin_imap" protocol="imap" classname="com.sun.mail.imap.IMAPStore" type="STORE"/>
  <protocolProviders xmi:id="builtin_smtps" protocol="smtps" classname="com.sun.mail.smtp.SMTPSSLTransport" type="TRANSPORT"/>
  <protocolProviders xmi:id="builtin_pop3s" protocol="pop3s" classname="com.sun.mail.pop3.POP3SSLStore" type="STORE"/>
  <protocolProviders xmi:id="builtin_imaps" protocol="imaps" classname="com.sun.mail.imap.IMAPSSLStore" type="STORE"/>
</resources.mail:MailProvider>
```

- Required fields:
  - Mail Provider Properties page: name, and at least one protocol provider
  - Mail Session Properties page: name, jndiName, outgoing server and protocol, and/or incoming server and protocol
- Special cases:
  - If you use the ACRCT tool, the password field will be encrypted. You cannot encrypt the password field if you do not use the ACRCT tool.

Mail provider settings for application clients:

Use this page to implement the JavaMail API and create mail sessions.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Mail Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

The name of the JavaMail resource provider.

Description:

An optional description for the resource provider.

Class Path:

Specifies a list of paths or JAR file names which together form the location for the resource provider classes.

Protocol:

Specifies the name of the protocol.

Classname:

Specifies the name of the class implementing the protocol. Leave this field blank if you want to use the default implementation.

Type:

This menu contains the following two values: `TRANSPORT` or `STORE`.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Mail session settings for application clients:

Use this page to configure mail session properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Mail Providers** > *mail provider instance*. Right-click **Mail Sessions** and click **New**. The following fields appear on the **General** tab:

Name:

Represents the administrative name of the JavaMail session object.

Description:

Provides an optional description for your administrative records.

JNDI Name:

The application client run time uses this field to retrieve configuration information.

Mail Transport Host:

Specifies the server to connect to when sending mail.

Mail Transport Protocol:

Specifies the transport protocol to use when sending mail.

Mail Transport User:

Specifies the user ID to use when the mail transport host requires authentication.

Mail Transport Password:

Specifies the password to use when the mail transport host requires authentication.

Enable strict Internet address parsing:

Specifies whether the recipient addresses must be parsed strictly in compliance with RFC 822, which is a specifications document issued by the Internet Architecture Board.

This setting is not generally used for most mail applications. RFC 822 syntax for parsing addresses effectively enforces a strict definition of a valid email address. If you select this setting, JavaMail will adhere to RFC 822 syntax and reject recipient addresses that do not parse into valid email addresses (as defined by the specification). If you do not select this setting, JavaMail will not adhere to RFC 822 syntax and will accept recipient addresses that do not comply with the specification. By default, this setting is deselected. You can view the RFC 822 specification at the following URL for the World Wide Web Consortium (W3C): http://www.w3.org/Protocols/rfc822/.

Re-Enter Password:

Confirms the password.

Mail From:

Specifies the mail originator.

Mail Store Host:

Specifies the mail account host (or "domain") name.

Mail Store User:

Specifies the user ID of the mail account.

Mail Store Password:

Specifies the password of the mail account.

Re-Enter Password:

Confirms the password.

Mail Store Protocol:

Specifies the protocol to be used when receiving mail.

Mail Debug:

When `true`, JavaMail interaction with mail servers, along with these mail session properties are printed to the `stdout` file.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring mail provider and mail session settings for application clients:

You can configure mail provider and mail session settings. This topic provides the required fields, special cases, and an example.

The purpose of this topic is to help you configure mail provider and mail session settings.

- Required fields:
  - Mail Provider Properties page: name, and at least one protocol provider
  - Mail Session Properties page: name, jndiName, outgoing server and protocol, and/or incoming server and protocol
- Special cases:
  - If you use the ACRCT tool, the password field will be encrypted. You cannot encrypt the password field if you do not use the ACRCT tool.
- Example:

```
<resources.mail:MailProvider xmi:id="builtin_mailprovider" name="Built-in Mail Provider" description="The built-in mail provider">
  <factories xmi:type="resources.mail:MailSession"
             xmi:id="MailSession_1207766754834" name="MailSession"
             jndiName="mail/session" description="Sample mail session" category="Sample"
             mailTransportHost="smtp.coldmail.com" mailTransportUser="transportUser"
             mailTransportPassword="{xor}Lz4sLChvLTs="
             mailFrom="smith@coldmail.com" mailStoreHost="imap.coldmail.com" mailStoreUser="storeUser"
             mailStorePassword="{xor}Lz4sLChvLTs="
             debug="true" strict="true"
             mailTransportProtocol="builtin_smtp" mailStoreProtocol="builtin_imap">
    <propertySet xmi:id="J2EEResourcePropertySet_1207766778585">
      <resourceProperties xmi:id="J2EEResourceProperty_1207766778585" name="key" type="java.lang.String" value="value" required="false"/>
    </propertySet>
```

```
  </factories>
  <protocolProviders xmi:id="builtin_smtp" protocol="smtp" classname="com.sun.mail.smtp.SMTPTransport" type="TRANSPORT"/>
  <protocolProviders xmi:id="builtin_pop3" protocol="pop3" classname="com.sun.mail.pop3.POP3Store" type="STORE"/>
  <protocolProviders xmi:id="builtin_imap" protocol="imap" classname="com.sun.mail.imap.IMAPStore" type="STORE"/>
  <protocolProviders xmi:id="builtin_smtps" protocol="smtps" classname="com.sun.mail.smtp.SMTPSSLTransport" type="TRANSPORT"/>
  <protocolProviders xmi:id="builtin_pop3s" protocol="pop3s" classname="com.sun.mail.pop3.POP3SSLStore" type="STORE"/>
  <protocolProviders xmi:id="builtin_imaps" protocol="imaps" classname="com.sun.mail.imap.IMAPSSLStore" type="STORE"/>
</resources.mail:MailProvider>
```

Configuring new mail sessions for application clients

You can use the Application Client Resource Configuration Tool (ACRCT) to configure new mail sessions for your application client.

Before you begin

During this task, you configure new mail sessions for your application client. The mail sessions are associated with the pre-configured default mail provider supplied by the product.

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file. The EAR file contents are displayed in a tree view.
2. Select the JAR file in which you want to configure the new JavaMail session.
3. Expand the JAR file to view its contents.
4. Click **Mail Providers > Mail Provider > Mail Sessions**. Complete one of the following actions:
   - Right click the Mail Sessions folder and select **New Factory**.
   - Click **Edit > New** on the menu bar.
5. Configure the Mail Session properties in the displayed fields.
6. Click **OK**.
7. Click **File > Save** on the menu bar to save your changes.

Configuring new URL providers for application clients

You can create URL providers and URLs for your client application using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create URL providers and URLs for your client application. In a separate administrative task, you must install the Java code for the required URL provider on the client machine on which the client application resides.

About this task

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new URL provider. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new URL provider from the tree.
4. Expand the JAR file to view the contents.
5. Click the folder called URL Providers. Complete one of the following actions:
   - Right click the folder and select **New**.
   - Click **Edit > New** on the menu bar.
6. Configure the URL provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File > Save** on the menu bar to save your changes.

Example

- Configuring URL and URL provider settings for application clients

This code example illustrates how to configure URL and URL provider settings for application clients:

```
<resources.url:URLProvider xmi:id="URLProvider_1" name="urlProvider:name"
description="urlProvider:description"
streamHandlerClassName="urlProvider:streamHandlerClass"
protocol="urlProvider:protocol">
<classpath>urlProvider:classpath</classpath>
<factories xmi:type="resources.url:URL" xmi:id="URL_1" name="urlFactory:name"
jndiName="urlFactory:jndiName" description="urlFactory:description"
spec="urlFactory:url">
<propertySet xmi:id="J2EEResourcePropertySet_18">
<resourceProperties xmi:id="J2EEResourceProperty_20" name="urlFactory:customName"
value="urlFactory:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_19">
<resourceProperties xmi:id="J2EEResourceProperty_21" name="urlProvider:customName"
value="urlProvider:customValue"/>
</propertySet>
</resources.url:URLProvider>
```

- Required fields:
  - URL Properties page: name, jndiName, url
  - URL Provider Properties page: name

URLs for application clients:

A *Uniform Resource Locator* (URL) is an identifier that points to an electronically accessible resource, such as a directory file on a machine in a network, or a document stored in a database.

URLs appear in the format *scheme*:*scheme_information*.

You can represent a *scheme* as `http`, `ftp`, `file`, or another term that identifies the type of resource and the mechanism by which you can access the resource.

In a web browser location or address box, a URL for a file available using HyperText Transfer Protocol (HTTP) starts with `http:`. An example is `http://www.ibm.com`. Files available using File Transfer Protocol (FTP) start with `ftp:`. Files available locally start with `file:`.

The *scheme_information* commonly identifies the Internet machine making a resource available, the path to that resource, and the resource name. The scheme_information for HTTP, FTP and File generally starts with two slashes (//), then provides the Internet address separated from the resource path name with one slash (/). For example,

`http://www.ibm.com/software/webservers/appserv/library.html`.

For HTTP and FTP, the path name ends in a slash when the URL points to a directory. In such cases, the server generally returns the default index for the directory.

URL providers for the Application Client Resource Configuration Tool:

A URL provider implements the function for a particular URL protocol, such as HyperText Transfer Protocol (HTTP). This provider, comprised of a pair of classes, extends the java.net.URLStreamHandler and java.net.URLConnection classes.

Configuring URL providers and sessions using the Application Client Resource Configuration Tool:

You can edit the configurations of URL providers and URLs to be used by your application clients using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of URL providers and URLs to be used by your application clients.

About this task

Procedure

1. Start the ACRCT.
2. Open an EAR file.
3. Locate the URL objects in the tree that displays. For example, if your file contains URL providers and URLs, expand **Resources > *application* > .jar > URL Providers > *url_provider_instance***
   where *url_provider_instance* is a particular URL provider.
4. If you expand the tree further, you will also see the URLs folders containing the URL instances for each URL provider instance.

URL settings for application clients:

Use this page to implement the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP).

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **URL Providers** > *URL provider instance*. Right-click **URLs** and click **New**. The following fields appear on the **General** tab.

This provider, comprised of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Name:

The administrative name for the URL.

Description:

This is an optional description of the URL for your administrative records.

JNDI Name:

The application client run time uses this field to retrieve configuration information.

URL:

A Uniform Resource Locator (URL) name that points to an Internet or intranet resource. For example: `http://www.ibm.com`.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

URL provider settings for application clients:

Use this page create new URL providers.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right click **URL Providers**, and click **New**. The following fields appear on the **General** tab.

A URL provider implements the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP). This provider, comprised of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Name:

Administrative name for the URL.

Description:

Optional description of the URL, for your administrative records.

Class Path:

A list of paths or JAR file names which together form the location for the resource provider classes.

Protocol:

Protocol supported by this stream handler. For example, `nntp`, `smtp`, `ftp`, and so on.

To use the default protocol, leave this field blank.

Stream handler class:

Fully qualified name of a User-defined Java class that extends the `java.net.URLStreamHandler` for a particular URL protocol, such as FTP.

To use the default stream handler, leave this field blank.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring URL and URL provider settings for application clients:

You can configure URL and URL provider settings. This topic provides the required fields and an example.

The purpose of this article is to help you to configure URL and URL provider settings.

- Required fields:
  - URL Properties page: name, jndiName, url

  – URL Provider Properties page: name
- Example:

```
<resources.url:URLProvider xmi:id="URLProvider_1" name="urlProvider:name"
description="urlProvider:description"
streamHandlerClassName="urlProvider:streamHandlerClass"
protocol="urlProvider:protocol">
<classpath>urlProvider:classpath</classpath>
<factories xmi:type="resources.url:URL" xmi:id="URL_1" name="urlFactory:name"
jndiName="urlFactory:jndiName" description="urlFactory:description"
spec="urlFactory:url">
<propertySet xmi:id="J2EEResourcePropertySet_18">
<resourceProperties xmi:id="J2EEResourceProperty_20" name="urlFactory:customName"
value="urlFactory:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_19">
<resourceProperties xmi:id="J2EEResourceProperty_21" name="urlProvider:customName"
value="urlProvider:customValue"/>
</propertySet>
</resources.url:URLProvider>
```

Configuring new URLs with the Application Client Resource Configuration Tool

You can use URLs for your client application using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create URLs for your client application.

About this task

Procedure

1. Click the URL provider for which you want to create a URL in the tree. Complete one of the following:
   - Configure a new URL provider.
   - Click an existing URL provider.
2. Expand the URL provider to view the URLs folder.
3. Click the URL folder. Complete one of the following actions:
   - Right click the folder and click **New**.
   - Click **Edit -> New** on the menu bar.
4. Configure the URL properties in the displayed fields.
5. Click **OK** when you finish.
6. Click **File > Save** in the menu bar to save your changes.

Configuring Java messaging client resources

To configure Java messaging client resources, you create new JMS provider configurations for your application client. The application client can use a messaging service through the Java Message Service APIs. A JMS provider provides two kinds of J2EE factories. One is a *JMS connection factory*, and the other is a *JMS destination factory*.

Before you begin

In a separate administrative task, install the Java Message Service (JMS) client on the client machine where the application client resides. The messaging product vendor must provide an implementation of the JMS client. For more information, see your messaging product documentation.

Attention: When completing this task, you can either create a new messaging provider, or you can use an existing one.

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new JMS provider. The EAR file contents are in the displayed tree view.
3. Select the JAR file in which you want to configure the new JMS provider from the tree.
4. Expand the JAR file to view its contents.
5. Optionally right-click **Messaging Providers** and select **New**, if you want to create and use a new messaging provider.
6. Configure the JMS provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File > Save**.

Asynchronous messaging in WebSphere Application Server using JMS:

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interface. The JMS interface provides a common way for Java programs (clients and Java Platform, Enterprise Edition (Java EE) applications) to create, send, receive, and read asynchronous requests as JMS messages.

This topic provides a generic overview of asynchronous messaging using the JMS support provided by WebSphere Application Server.

The base support for asynchronous messaging using the JMS API provides the common set of JMS interfaces and associated semantics that define how a JMS client can access the facilities of a JMS provider. This support enables WebSphere product Java EE applications, as JMS clients, to exchange messages asynchronously with other JMS clients, by using JMS destinations (queues or topics). A Java EE application can use JMS queue destinations for point-to-point messaging and JMS topic destinations for publish and subscribe messaging. A Java EE application can explicitly poll for messages on a destination, and then retrieve messages for processing by business logic beans (enterprise beans).

With the base JMS and XA support, the Java EE application uses standard JMS calls to process messages, including any responses or outbound messaging. An enterprise bean can handle responses acting as a sender bean, or within the enterprise bean that receives the incoming messages. Optionally, this process can use two-phase commit within the scope of a transaction. This level of function for asynchronous messaging is called *bean-managed messaging*, and gives an enterprise bean complete control over the messaging infrastructure, for example, connection and session pool management. The common container has no role in bean-managed messaging.

WebSphere Application Server also supports automatic asynchronous messaging using message-driven beans (a type of enterprise bean defined in the Enterprise JavaBeans (EJB) 2.0 specification) and JMS listeners (part of the JMS application server facilities). Messages are automatically retrieved from JMS destinations, optionally within a transaction, then sent to the message-driven bean in a Java EE application, without the application having to explicitly poll JMS destinations.

Java Message Service providers for clients:

Client applications can use messaging resources from three main types of Java Message Service (JMS) providers in WebSphere Application Server: The WebSphere Application Server default messaging provider (which uses service integration as the provider), the WebSphere MQ messaging provider (which uses your WebSphere MQ system as the provider) and third-party messaging providers (which use another company's product as the provider).

IBM WebSphere Application Server supports asynchronous messaging through the use of a JMS provider and its related messaging system. JMS providers must conform to the JMS specification version 1.1. To

use message-driven beans the JMS provider must support the optional Application Server Facility (ASF) function defined within that specification, or support an inbound resource adapter as defined in the JCA specification version 1.5.

Default messaging provider
If you mainly want to use messaging between applications in WebSphere Application Server, perhaps with some interaction with a WebSphere MQ system, the default messaging provider is the natural choice. This provider is based on service integration technologies and is fully integrated with the WebSphere Application Server runtime environment.

WebSphere MQ messaging provider
If your business also uses WebSphere MQ, and you want to integrate WebSphere Application Server messaging applications into a predominately WebSphere MQ network, choose the WebSphere MQ messaging provider, which allows you to define resources for connecting to any queue manager on the WebSphere MQ network.

Third-party messaging provider
You can configure any third-party messaging provider that supports the JMS Version 1.1 unified connection factory. You might want to do this, for example, because of existing investments.

WebSphere applications can use messaging resources provided by any of these JMS providers. However the choice of provider is most often dictated by requirements to use or integrate with an existing messaging system. For example, you may already have a messaging infrastructure based on WebSphere MQ. In this case you may either connect directly using the included support for WebSphere MQ as a JMS provider, or configure a service integration bus with links to a WebSphere MQ network and then access the bus through the default messaging provider.

Configuring new JMS providers with the Application Client Resource Configuration Tool:

You can create new Java Message Service (JMS) provider configurations for the Application Client. The Application Client makes use of a messaging service through the JMS interfaces.

About this task

During this task, you create new Java Message Service (JMS) provider configurations for the Application Client. The Application Client makes use of a messaging service through the JMS interfaces. A JMS provider provides two kinds of Java Platform, Enterprise Edition (Java EE) resources. One is a JMS connection factory, and the other is a JMS destination.

In a separate administrative task, you must install the JMS client on the client machine where your particular application client resides. The messaging product vendor must provide an implementation of the JMS client. For more information, see your messaging product documentation.

Procedure

1. Start the Application Client Resource Configuration Tool and open the EAR file for which you want to configure the new JMS provider. The EAR file contents are displayed in a tree view.
2. From the tree, select the JAR file in which you want to configure the new JMS provider.
3. Expand the JAR file to view its contents.
4. Right-click **Messaging Providers**. Complete one of the following actions:
   - Right click the folder and select **New**.
   - On the menu bar, click **Edit > New**.
5. In the resulting property dialog, configure the JMS provider properties.
6. Click **OK** when finished.
7. Click **File > Save** on the menu bar to save your changes.

Example

The following code example illustrates how to configure JMS Provider, JMS Connection Factory and JMS Destination settings for application clients.

```
<resources.jms:JMSProvider xmi:id="JMSProvider_3" name="genericJMSProvider:name"
description="genericJMSProvider:description"
externalInitialContextFactory="genericJMSProvider:contextFactoryClass"
externalProviderURL="genericJMSProvider:providerUrl">
<classpath>genericJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms:GenericJMSDestination"
xmi:id="GenericJMSDestination_1" name="jmsDestination:name"
jndiName="jmsDestination:jndiName" description="jmsDestination:description"
externalJNDIName="jmsDestination:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_15">
<resourceProperties xmi:id="J2EEResourceProperty_17" name="jmsDestination:customName"
value="jmsDestination:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms:GenericJMSConnectionFactory"
xmi:id="GenericJMSConnectionFactory_1" name="jmsCF:name" jndiName="jmsCF:jndiName"
description="jmsCF:description" userID="jmsCF:user" password="{xor}NTIsHBllMT4yOg=="
externalJNDIName="jmsCF:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_16">
<resourceProperties xmi:id="J2EEResourceProperty_18" name="jmsCF:customName"
value="jmsCF:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_17">
<resourceProperties xmi:id="J2EEResourceProperty_19"
name="genericJMSProvider:customName" value="genericJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Required fields include:
- JMS Provider Properties page: name, and at least one protocol provider
- JMS Connection Factory Properties page: name, jndiName, destination type
- JMS Destination Properties page: name, jndiName, destination type

Special cases:
- The destination type must be QUEUE, or TOPIC.

JMS provider settings for application clients:

Use this page to configure properties of the Java Message Service (JMS) provider, if you want to use a JMS provider other than the default messaging provider or the WebSphere MQ as a JMS provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right click **Messaging Providers**, and click **New**. The following fields appear on the **General** tab.

Name:

The name by which the JMS provider is known for administrative purposes.

Information	**Value**
Data type	String

Description:

A description of the JMS provider, for administrative purposes.

Information	Value
Data type	String

Class Path:

A list of paths or `.jar` file names which together form the location for the resource provider classes.

Context factory class:

The Java class name of the initial context factory for the JMS provider.

For example, for an LDAP service provider the value has the form: `com.sun.jndi.ldap.LdapCtxFactory`.

Information	Value
Data type	String

Provider URL:

The JMS provider URL for external JNDI lookups.

For example, an LDAP URL for a JMS provider has the form: `ldap://hostname.company.com/contextName`.

Information	Value
Data type	String

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Default Provider connection factory settings:

Use this panel to view or change the configuration properties of the selected JMS connection factory for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server. These configuration properties control how connections are created between the JMS provider and the service integration bus that it uses

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Connection Factories** and click **New**. The following fields appear on the **General** tab.

Settings that have a default value display the appropriate value. Any settings that have fixed values have a drop down menu.

Name:

The name of the connection factory.

Information	Value
Data type	String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The JNDI name that is used to match this Resource Adapter connection factory definition to the deployment descriptor. This entry is a `resource-ref` name.

Information	Value
Data type	String

User Name:

The **User Name** used with the **Password** property for connecting to an application.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly. If a user name and password are specified, then an authentication alias is created for the factory where the password is encrypted.

Information	Value
Data type	String

Password:

The password used to authenticate connection to an application.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

Information	Value
Data type	String

Re-Enter Password:

Confirms the password.

Bus Name:

The name of the bus to which the connection factory connects.

Information	Value
Data type	String

Client Identifier:

The name of the client. Required for durable topic subscriptions.

Information	Value
Data type	String

Nonpersistent Messaging Reliability:

The reliability applied to nonpersistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus** destination. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Information	Value
Default	ReliablePersistent
Range	**None** There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded. **Best effort nonpersistent** Messages are never written to disk, and are thrown away if memory cache overruns. **Express nonpersistent** Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts. **Reliable nonpersistent** Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions. **Reliable persistent** Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions. **Assured persistent** Highest degree of reliability where assured message delivery is supported. **As Bus destination** Use the delivery option configured for the bus destination.

Persistent Message Reliability:

The reliability applied to persistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus destination**. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Information	Value
Default	ReliablePersistent
Range	**None** There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded. **Best effort nonpersistent** Messages are never written to disk, and are thrown away if memory cache overruns. **Express nonpersistent** Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts. **Reliable nonpersistent** Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions. **Reliable persistent** Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions. **Assured persistent** Highest degree of reliability where assured message delivery is supported. **As Bus destination** Use the delivery option configured for the bus destination.

Durable Subscription Home:

The name of the durable subscription home.

Information	Value
Data type	String

Share durable subscriptions:

Controls whether or not durable subscriptions are shared across connections with members of a server cluster.

Normally, only one session at a time can have a TopicSubscriber for a particular durable subscription. This property enables you to override this behavior, to enable a durable subscription to have multiple simultaneous consumers.

Information	Value
Data type	Selection list
Default	In cluster

Information	Value
Range	**In cluster** Allows sharing of durable subscriptions when connections are made from within a server cluster. **Always shared** Durable subscriptions can be shared across connections. **Never shared** Durable subscriptions are never shared across connections.

Read Ahead:

Controls the read-ahead optimization during message delivery.

Information	Value
Default	Default
Range	Default, AlwaysOn and AlwaysOff

Target:

The name of the Workload Manager target group containing the messaging engine.

Information	Value
Data type	String

Target Type:

The type of Workload Manager target group that contains the messaging engine.

Information	Value
Default	BusMember
Range	BusMember, Custom, ME

Target Significance:

The priority of significance for the target specified.

Information	Value
Default	Preferred
Range	Preferred, Required

Target Inbound Transport Chain:

The name of the protocol that resolves to a group of messaging engines.

Information	Value
Data type	String

Provider Endpoints:

The list of comma separated endpoints used to connect to a bootstrap server.

Type a comma-separated list of endpoint triplets with the syntax: host:port:protocol.

Information	Value
Example	`merlin:7276:BootstrapBasicMessaging,Gandalf:5557:BootstrapSecureMessaging` where `BootstrapBasicMessaging` corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).
Default	• If the host name is not specified, then the default `localhost` is used as a default value. • If the port number is not specified, then `7276` is used as a default value. • If the chain name is not specified, a predefined chain, such as `BootstrapBasicMessaging`, is used as a default value.

Connection Proximity:

The proximity that the messaging engine should have to the requester.

Information	Value
Default	Bus
Range	Bus, Host, Cluster, Server

Temporary Queue Name Prefix:

The prefix to apply to the names of temporary queues. This name is a maximum of 12 characters.

Information	Value
Data type	String

Temporary Topic Name Prefix:

The prefix to apply to the names of temporary topics. This name is a maximum of 12 characters.

Information	Value
Data type	String

Default Provider queue connection factory settings:

Use this panel to view or change the configuration properties of the selected JMS queue connection factory for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server. These configuration properties control how connections are created between the JMS provider and the service integration bus that it uses

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Queue Connection Factories** and click **New**. The following fields appear on the **General** tab.

Settings that have a default value display the appropriate value. Any settings that have fixed values have a drop down menu.

Name:

The name of the queue connection factory.

Information	Value
Data type	String

Description:

A description of this queue connection factory for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The JNDI name that is used to match this queue connection factory definition to the deployment descriptor. This entry is a `resource-ref` name.

Information	Value
Data type	String

User Name:

The **User Name** used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly. If this field is used, then the Properties field `UserName` is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User Name** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly. If a user name and password are specified, then an authentication alias is created for the factory where the password is encrypted.

Information	Value
Data type	String

Password:

The password used to create an encrypted. If you complete this field, then the Password field in the Properties box is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

Information	Value
Data type	String

Re-Enter Password:

Confirms the password.

Bus Name:

The name of the bus to which the queue connection factory connects.

Information	Value
Data type	String

Client Identifier:

The client identifier. Required for durable topic subscriptions.

Information	Value
Data type	String

Nonpersistent Messaging Reliability:

The reliability applied to nonpersistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus** destination. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Information	Value
Default	ReliablePersistent
Range	**None** There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded. **Best effort nonpersistent** Messages are never written to disk, and are thrown away if memory cache overruns. **Express nonpersistent** Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts. **Reliable nonpersistent** Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions. **Reliable persistent** Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions. **Assured persistent** Highest degree of reliability where assured message delivery is supported. **As Bus destination** Use the delivery option configured for the bus destination.

Persistent Message Reliability:

The reliability applied to persistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus destination**. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Information	Value
Default	ReliablePersistent
Range	**None** There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded. **Best effort nonpersistent** Messages are never written to disk, and are thrown away if memory cache overruns. **Express nonpersistent** Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts. **Reliable nonpersistent** Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions. **Reliable persistent** Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions. **Assured persistent** Highest degree of reliability where assured message delivery is supported. **As Bus destination** Use the delivery option configured for the bus destination.

Read Ahead:

Controls the read-ahead optimization during message delivery.

Information	Value
Default	Default
Range	Default, AlwaysOn and AlwaysOff

Target:

The name of the Workload Manager target group containing the messaging engine.

Information	Value
Data type	String

Target Type:

The type of Workload Manager target group that contains the messaging engine.

Information	Value
Default	BusMember
Range	BusMember, Custom, Destination, ME

Target Significance:

The priority of significance for the target specified.

Information	Value
Default	Preferred
Range	Preferred, Required

Target Inbound Transport Chain:

The name of the protocol that resolves to a group of messaging engines.

Information	Value
Data type	String

Provider Endpoints:

The list of comma separated endpoints used to connect to a bootstrap server.

Type a comma-separated list of endpoint triplets with the syntax: host:port:protocol.

Information	Value
Example	`localhost:7777:BootstrapBasicMessaging` where `BootstrapBasicMessaging` corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).
Default	• If the host name is not specified, then the default `localhost` is used as a default value. • If the port number is not specified, then `7276` is used as a default value. • If the chain name is not specified, a predefined chain, such as `BootstrapBasicMessaging`, is used as a default value.

Connection Proximity:

The proximity that the messaging engine should have to the requester.

Information	Value
Default	Bus, Cluster, Server
Range	Bus, Host

Temporary Queue Name Prefix:

The prefix to apply to the names of temporary queues. This name is a maximum of 12 characters.

Information	Value
Data type	String

Default Provider topic connection factory settings:

Use this panel to view or change the configuration properties of the selected JMS topic connection factory for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server. These configuration properties control how connections are created between the JMS provider and the service integration bus that it uses.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Topic Connection Factories** and click **New**. The following fields appear on the **General** tab.

Settings that have a default value display that appropriate value. Any settings that have fixed values have a drop down menu.

Name:

The name of the topic connection factory.

Information	**Value**
Data type	String

Description:

A description of this topic connection factory for administrative purposes within IBM WebSphere Application Server.

Information	**Value**
Data type	String

JNDI Name:

The JNDI name that is used to match this topic connection factory definition to the deployment descriptor. This entry is a `resource-ref` name.

Information	**Value**
Data type	String

User Name:

The **User Name** used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly. If this field is used, then the Properties field `UserName` is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

The connection factory **User Name** and **Password** properties are used if the calling application does not provide a `userid` and `password` explicitly. If a user name and password are specified, then an authentication alias is created for the factory where the password is encrypted.

Information	**Value**
Data type	String

Password:

The password used to create an encrypted. If you complete this field, then the Password field in the Properties box is ignored.

If you specify a value for the **User Name** property, you must also specify a value for the **Password** property.

Information	Value
Data type	String

Re-Enter Password:

Confirms the password.

Bus Name:

The name of the bus to which the topic connection factory connects.

Information	Value
Data type	String

Client Identifier:

The name of the client. This field is required for durable topic subscriptions.

Information	Value
Data type	String

Nonpersistent Messaging Reliability:

The reliability applied to nonpersistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus** destination. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Information	Value
Default	ReliablePersistent

Information	Value
Range	**None** There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded. **Best effort nonpersistent** Messages are never written to disk, and are thrown away if memory cache overruns. **Express nonpersistent** Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts. **Reliable nonpersistent** Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions. **Reliable persistent** Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions. **Assured persistent** Highest degree of reliability where assured message delivery is supported. **As Bus destination** Use the delivery option configured for the bus destination.

Persistent Message Reliability:

The reliability applied to persistent JMS messages sent using this connection factory.

If you want different reliability delivery options for individual JMS destinations, you can set this property to **As bus destination**. The reliability is then defined by the Reliability property of the bus destination to which the JMS destination is assigned.

Information	Value
Default	ReliablePersistent

Information	Value
Range	**None** There is no message reliability for nonpersistent messages. If a nonpersistent message cannot be delivered, it is discarded. **Best effort nonpersistent** Messages are never written to disk, and are thrown away if memory cache overruns. **Express nonpersistent** Messages are written asynchronously to persistent storage if memory cache overruns, but are not kept over server restarts. **Reliable nonpersistent** Messages can be lost if a messaging engine fails, and can be lost under normal operating conditions. **Reliable persistent** Messages can be lost if a messaging engine fails, but are not lost under normal operating conditions. **Assured persistent** Highest degree of reliability where assured message delivery is supported. **As Bus destination** Use the delivery option configured for the bus destination.

Durable Subscription Home:

The name of the durable subscription home.

Information	Value
Data type	String

Share durable subscriptions:

Controls whether or not durable subscriptions are shared across connections with members of a server cluster.

Normally, only one session at a time can have a TopicSubscriber for a particular durable subscription. This property enables you to override this behavior, to enable a durable subscription to have multiple simultaneous consumers.

Information	Value
Data type	Selection list
Default	In cluster

Information	Value
Range	**In cluster** Allows sharing of durable subscriptions when connections are made from within a server cluster. **Always shared** Durable subscriptions can be shared across connections. **Never shared** Durable subscriptions are never shared across connections.

Read Ahead:

Controls the read-ahead optimization during message delivery.

Information	Value
Default	Default
Range	Default, AlwaysOn and AlwaysOff

Target:

The name of the Workload Manager target group containing the messaging engine.

Information	Value
Data type	String

Target Type:

The type of Workload Manager target group that contains the messaging engine.

Information	Value
Default	BusMember
Range	BusMember, Custom, ME

Target Significance:

The priority of significance for the target specified.

Information	Value
Default	Preferred
Range	Preferred, Required

Target Inbound Transport Chain:

The name of the protocol that resolves to a group of messaging engines.

Information	Value
Data type	String

Provider Endpoints:

The list of comma separated endpoints used to connect to a bootstrap server.

Type a comma-separated list of endpoint triplets with the syntax: host:port:protocol.

Information	Value
Example	`localhost:7777:BootstrapBasicMessaging` where `BootstrapBasicMessaging` corresponds to the remote protocol InboundBasicMessaging (JFAP-TCP/IP).
Default	• If the host name is not specified, then the default `localhost` is used as a default value. • If the port number is not specified, then `7276` is used as a default value. • If the chain name is not specified, a predefined chain, such as `BootstrapBasicMessaging`, is used as a default value.

Connection Proximity:

The proximity that the messaging engine should have to the requester.

Information	Value
Default	Bus
Range	Bus, Host, Cluster, Server

Temporary Topic Name Prefix:

The prefix to apply to the names of temporary topics. This name is a maximum of 12 characters.

Information	Value
Data type	String

Default Provider queue destination settings:

Use this panel to view or change the configuration properties of the selected JMS queue destination for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Queue Destinations**. Click **New**. The following fields appear on the **General** tab.

Name:

The name of the queue destination factory. You must complete this field.

Information	Value
Data type	String

Description:

A description of this queue destination for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The JNDI name used to match this definition to a deployment descriptor `resource-env-ref` name.

Information	Value
Data type	String

Queue Name:

The name of the queue.

Information	Value
Data type	String

Delivery Mode:

The delivery mode for messages sent to this destination.

Information	Value
Data type	String
Range	Application, Persistent or NonPersistent
Default	Application

Time to Live:

The default length of time from its dispatch time that a message sent to this destination should be retained by the system, where **0** indicates that time to live value does not expire. Value from the producer is used if the Time to Live field is not completed.

Information	Value
Data type	Integer
Units	Milliseconds

Priority:

The priority for messages sent to this destination. The value from the producer is used if not completed.

Information	Value
Data type	Integer
Range	0 to 9 with **0** as the lowest priority and **9** as the highest priority

Read Ahead:

Used to control read-ahead optimization during message delivery.

Information	Value
Data type	String
Range	AsConnection, AlwaysOn and AlwaysOff

Information	Value
Default	AsConnection

Default Provider topic destination settings:

Use this panel to view or change the configuration properties of the selected JMS topic destination for use with the internal product Java Message Service (JMS) provider that is installed with WebSphere Application Server.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **Default Provider**. Right-click **Topic Destinations**, and click **New**. The following fields appear on the **General** tab.

Name:

The name of the topic destination entry.

Information	Value
Data type	String

Description:

A description of the entry.

Information	Value
Data type	String

JNDI Name:

The JNDI name used to match this definition to a deployment descriptor `resource-env-ref` name.

Information	Value
Data type	String

Topic Space:

The name of the topic space. This field is required.

Information	Value
Data type	String
Default	DEFAULT_TOPIC_SPACE

Topic Name:

The name of the topic. This field is required.

Information	Value
Data type	String

Delivery Mode:

The default mode for messages sent to this destination.

Information	Value
Data type	String
Range	Application, Persistent or NonPersistent
Default	Application

Time to Live:

The default length of time from its dispatch time that a message sent to this destination should be retained by the system, where **0** indicates that time to live value does not expire. Value from the producer is used if not completed.

Information	Value
Data type	Long
Units	Milliseconds

Priority:

The priority for messages sent to this destination. Value from producer is used if not completed.

Information	Value
Data type	Integer
Range	0 to 9 with **0** as the lowest priority and **9** as the highest priority

Read Ahead:

Used to control read-ahead optimization during message delivery.

Information	Value
Data type	String
Range	AsConnection, AlwaysOn and AlwaysOff
Default	AsConnection

WebSphere MQ Provider queue connection factory settings for application clients:

Use this panel to view or change the configuration properties of the selected queue connection factory for use with the WebSphere MQ Java Message Service (JMS) provider. These configuration properties control how connections are created between the JMS provider and WebSphere MQ.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file and click **Messaging Providers** > **WebSphere MQ Provider**. Right click **Queue Connection Factories**, and click **New**. The following fields are displayed on the **General** tab.

Note:

- The property values that you specify must match the values that you specified when configuring WebSphere MQ for JMS resources. For more information about configuring WebSphere MQ for JMS resources, see the *Using Java* section of the WebSphere MQ information center.
- In WebSphere MQ, names can have a maximum of 48 characters, except for channels which have a maximum of 20 characters.

A queue connection factory for the JMS provider has the following properties.

Name:

The name by which this queue connection factory is known for administrative purposes within WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Information	Value
Data type	String

Description:

A description of this connection factory for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String
Default	Null

JNDI Name:

The application client run time uses this field to retrieve configuration information.

User ID:

The user ID used, with the password property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the user ID property, you must also specify a value for the password property.

The connection factory user ID and password properties are used if the calling application does not provide a user ID and password explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the user ID and password to the JMS server.

Information	Value
Data type	String

Password:

The password used, with the user ID property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the user ID property, you must also specify a value for the password property.

Information	Value
Data type	String
Default	Null

Re-Enter Password:

Confirms the password.

Queue Manager:

The name of the WebSphere MQ queue manager for this connection factory.

Connections created by this factory connect to that queue manager.

Information	Value
Data type	String

Enter Hostname and Port Information:

This radio button is selected by default and, if selected, enables the host and port properties and disables the connection name list property.

Information	Value
Data type	Radio button
Default	Selected

Host:

The name of the host on which the WebSphere MQ queue manager runs for client connection only.

Information	Value
Data type	String
Default	Null
Range	A valid TCP/IP host name

Port:

The TCP/IP port number used for connection to the WebSphere MQ queue manager, for client connection only.

This port must be configured on the WebSphere MQ queue manager.

Information	Value
Data type	Integer
Default	Null
Range	A valid TCP/IP port number, configured on the WebSphere MQ queue manager.

Enter Connection Name List Information:

If selected, this radio button enables the connection name list property and disables the host and port name properties. Select this radio button if you want to connect to a multi-instance queue manager.

Information	Value
Data type	Radio button
Default	Cleared

Connection Name List:

A comma-separated list of host and port information which can be used to connect to a multi-instance queue manager.

The format of the list is:

```
host[(port)],[host[(port)]]
```

where port is optional and defaults to 1414 if it is not set. For example:

```
hostname1,hostname2(1415)
```

For further information about multi-instance queue managers, see the WebSphere MQ information center.

This property must only be used for connecting to a multi-instance queue manager. It must not be used for connecting to a list of distinct queue managers as that can result in transaction integrity issues.

Channel:

The name of the channel used for connection to the WebSphere MQ queue manager, for client connection only.

Information	**Value**
Data type	String
Default	Null
Range	1 through 20 ASCII characters

Transport type:

Specifies whether the WebSphere MQ client connection or JNDI bindings are used for connection to the WebSphere MQ queue manager. The external JMS provider controls the communication protocols between JMS clients and JMS servers. Tune the transport type when you are using non-ASF nonpersistent, nondurable, nontransactional messaging or when you want to satisfy security issues and the client is local to the queue manager node.

Information	**Value**
Data type	Enum
Units	Not applicable
Default	BINDINGS
Range	**BINDINGS** JNDI bindings are used to connect to the queue manager. BINDINGS is a shared memory protocol and can only be used when the queue manager is on the same node as the JMS client and poses security risks that must be addressed through the use of EJB roles. **CLIENT** WebSphere MQ client connection is used to connect to the queue manager. CLIENT is a typical TCP-based protocol. **DIRECT** For WebSphere MQ Event Broker using DIRECT mode. DIRECT is a lightweight sockets protocol used in nontransactional, nondurable, and nonpersistent Publish/Subscribe messaging. DIRECT only works for clients and message-driven beans using the non-ASF protocol. **QUEUED** QUEUED is a standard TCP protocol.

Information	Value
Recommended	**Queue connection factory transport type** BINDINGS is faster by 30% or more, but it requires correctly set up EJB roles to guarantee security. If you have security concerns and need to use CLIENT then you should make appropriate use of SSL to secure the connection to the queue manager. **Topic connection factory transport type** DIRECT is the fastest type and must be used where possible. Use BINDINGS when you want to satisfy additional security tasks and the queue manager is local to the JMS client. QUEUED is the fallback for all other cases. WebSphere MQ 5.3 before CSD2 with the DIRECT setting can lose messages when used with message-driven beans and under load. This loss also happens with client-side applications unless the broker maxClientQueueSize is set to 0. You can set this value to 0 with the command: `#wempschangeproperties WAS_nodeName_server1 -e default -o DynamicSubscriptionEngine -n maxClientQueueSize -v 0 -x executionGroupUUID` where executionGroupUUID can be found by starting the broker and looking in the Event Log/Applications for event 2201. This value is usually ffffffff-0000-0000-000000000000. **Note:** The WebSphere MQ 5.3 JMS cannot be used within WebSphere Application Server Version 6.1 because WebSphere Application Server Version 6.1 has a Java 5 runtime. Therefore, cross-memory connections cannot be established with WebSphere MQ 5.3 queue managers. This can result in a performance degradation if you were previously using WebSphere MQ 5.3 and BINDINGS for your connections and move to CLIENT network connections in migrating to WebSphere Application Server Version 6.1. If you are using WebSphere MQ 5.3 for z/OS, you might also need to install an additional feature pack. When running on 64-bit z/OS, the Transport type must be set to CLIENT because the 64-bit WebSphere MQ z/OS is not currently available, and BINDINGS mode cannot be used to connect to 31-bit WebSphere MQ z/OS. You might also need to purchase an additional WebSphere MQ feature pack for this support.

Client ID:

The JMS client identifier used for connections to the WebSphere MQ queue manager.

Information	Value
Data type	String

CCSID:

The coded character set identifier for use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Information	Value
Data type	String

For more information about supported CCSIDs, and about converting between message data from one coded character set to another, see the *System Administration* and *Application Programming Reference* sections of the WebSphere MQ information center.

Message Retention:

Select this check box to specify that unwanted messages are to be left on the queue. Otherwise, unwanted messages are handled according to their disposition options.

Information	Value
Data type	Enum
Units	Not applicable
Default	Cleared
Range	**Selected** Unwanted messages are left on the queue. **Cleared** Unwanted messages are handled according to their disposition options.

Temporary model:

The name of the model definition used to create temporary connection factories if a connection factory does not already exist.

Information	Value
Data type	String
Range	1 through 48 ASCII characters

Temporary queue prefix:

The prefix used for dynamic queue naming.

Information	Value
Data type	String

Fail if quiesce:

Specifies whether applications return from a method call if the queue manager has entered a controlled failure.

Information	Value
Data type	Check box
Default	Selected

Local Server Address:

Specifies the local server address.

Information	Value
Data type	String

Polling Interval:

Specifies the interval, in milliseconds, between scans of all receivers during asynchronous message delivery

Information	Value
Data type	Integer

Information	Value
Units	Milliseconds
Default	5000

Rescan interval:

Specifies the interval in milliseconds between which a topic is scanned to look for messages that have been added to a topic out of order.

This interval controls the scanning for messages that have been added to a topic out of order with respect to a WebSphere MQ browse cursor.

Information	Value
Data type	Integer
Units	Milliseconds
Default	5000

SSL cipher suite:

Specifies the cipher suite to use for SSL connection to WebSphere MQ.

Set this property to a valid cipher suite provided by your JSSE provider. The value must match the CipherSpec specified on the SVRCONN channel as the **Channel** property.

You must set this property, if you set the **SSL Peer Name** property.

SSL certificate store:

Specifies a list of zero or more Certificate Revocation List (CRL) servers used to check for SSL certificate revocation. If you specify a value for this property, you must use WebSphere MQ JVM at Java 2 version 1.4.

The value is a space-delimited list of entries of the form:
`ldap://hostname:[port]`

A single slash (/) follows this value. If *port* is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers. For more information about CRL security, see the information about “Working with Certificate Revocation Lists” in the *Security* section of the WebSphere MQ information center.

SSL peer name:

For SSL, a *distinguished name* skeleton that must match the name provided by the WebSphere MQ queue manager. The distinguished name is used to check the identifying certificate presented by the server at connection time.

If this property is not set, such certificate checking is performed.

The SSL peer name property is ignored if **SSL Cipher Suite** property is not specified.

This property is a list of attribute name and value pairs separated by commas or semicolons. For example:
`CN=QMGR.*, OU=IBM, OU=WEBSPHERE`

The example given checks the identifying certificate presented by the server at connect-time. For the connection to succeed, the certificate must have a Common Name beginning QMGR., and must have at least two Organizational Unit names, the first of which is IBM and the second WEBSPHERE. Checking is not case-sensitive.

For more details about distinguished names and their use with WebSphere MQ, see the information about "Distinguished Names" in the WebSphere MQ information center.

Connection pool:

Specifies an optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the JMS provider to improve performance. This connection pooling is independent from any WebSphere MQ connection pooling. You must configure the connection and session pool properties appropriately for your applications, otherwise you might not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Information	**Value**
Data type	Check box
Default	Selected

Client reconnect options:

Specifies whether a client mode connection reconnects automatically, or not, in the event of a communications or queue manager failure. This property is ignored unless the connection factory is being used in a thin or managed client environment.

Information	**Value**
Data type	Drop-down list
Default	DISABLED
Range	**DISABLED** The client reconnection does not automatically occur. **ASDEF** The value from the DefRecon attribute from the channels stanza of the client configuration file is used. If there is no DefRecon value specified then this setting has the same effect as a value of DISABLED. **RECONNECT** Reconnection occurs to any queue manager consistent with the value of the queue manager attribute, which might be a different queue manager from that to which the connection was originally connected. **QMGR** Reconnection only occurs to the queue manager to which the connection was originally connected.

For more information about automatic client reconnection, see the WebSphere MQ information center.

Client reconnect timeout:

The maximum number of seconds that a client mode connection spends attempting to automatically reconnect to a queue manager after a communications or queue manager failure. This parameter is ignored unless the connection factory is being used in a thin or managed client environment. Whether this parameter is used or not depends on the value of the client reconnect options parameter.

Information	Value
Data type	Integer
Units	Seconds
Default	1800
Range	A value greater than zero and up to 2147483647

For more information about automatic client reconnection, see the WebSphere MQ information center.

WebSphere MQ Provider topic connection factory settings for application clients:

Use this panel to view or change the configuration properties of the selected topic connection factory for use with the WebSphere MQ Java Message Service (JMS) provider. These configuration properties control how connections are created between the JMS provider and WebSphere MQ.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **WebSphere MQ Provider**. Right-click **Topic Connection Factories** and click **New**.

Note:

- The property values that you specify must match the values that you specified when configuring WebSphere MQ product JMS resources. For more information about configuring WebSphere MQ JMS resources, see the *Using Java* section of the WebSphere MQ information center.
- In WebSphere MQ, names can have a maximum of 48 characters, except for channels which have a maximum of 20 characters.

MA0C broker: When creating a WebSphere Application Server Version 6 topic connection factory for the MA0C broker, consider the following attribute values:

BrokerControlQueue
This value is fixed at SYSTEM.BROKER.CONTROL.QUEUE for the MA0C broker and is the queue the broker reads from.

BrokerVersion
Set this value to BASIC for the MA0C broker.

ClientID
Set this value to whatever you like for the MA0C broker (the value is string and is merely an identifier for your client application).

XA Enabled
Set this value to TRUE or FALSE for the MA0C broker (the setting you use is a performance enhancement flag - you probably want to set this value to 'true' most of the time).

BrokerMessage Selection
This value is fixed at CLIENT for the MA0C broker because the broker relies on client side message selection.

Direct Broker Authorization Type
This value is not required by the MA0C broker.

A topic connection factory for the WebSphere MQ JMS provider has the following properties.

Name:

The name by which this topic connection factory is known for administrative purposes within WebSphere Application Server. The name must be unique within the JMS provider.

Information	Value
Data type	String

Description:

A description of this topic connection factory for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The Java Naming and Directory Interface (JNDI) name that is used to bind the topic connection factory into the application server name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 45 ASCII characters

User ID:

The user ID used, with the password property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the user ID property, you must also specify a value for the password property.

The connection factory user ID and password properties are used if the calling application does not provide a user ID and password explicitly, for example, if the calling application uses the method createTopicConnection(). The JMS client flows the user ID and password to the JMS server.

Information	Value
Data type	String

Password:

The password used, with the user ID property, for authentication if the calling application does not provide a user ID and password explicitly.

If you specify a value for the user ID property, you must also specify a value for the password property.

Information	Value
Data type	String

Re-Enter Password:

Confirms the password.

Queue Manager:

The name of the WebSphere MQ queue manager for this connection factory. Connections created by this connection factory connect to this queue manager.

Information	Value
Data type	String

Enter Hostname and Port Information:

This radio button is selected by default and, if selected, enables the host and port properties and disables the connection name list property.

Information	Value
Data type	Radio button
Default	Selected

Host:

The name of the host on which the WebSphere MQ queue manager runs for client connections only.

Information	Value
Data type	String
Range	A valid TCP/IP host name

Port:

The TCP/IP port number used for connection to the WebSphere MQ queue manager, for client connection only.

This port must be configured on the WebSphere MQ queue manager.

Information	Value
Data type	Integer
Range	A valid TCP/IP port number, configured on the WebSphere MQ queue manager.

Enter Connection Name List Information:

If selected, this radio button enables the connection name list property and disables the host and port name properties. Select this radio button if you want to connect to a multi-instance queue manager.

Information	Value
Data type	Radio button
Default	Cleared

Connection Name List:

A comma-separated list of host and port information which can be used to connect to a multi-instance queue manager.

The format of the list is:

```
host[(port)],[host[(port)]]
```

where port is optional and defaults to 1414 if it is not set. For example:

```
hostname1,hostname2(1415)
```

For further information about multi-instance queue managers, see the WebSphere MQ information center.

This property must only be used for connecting to a multi-instance queue manager. It must not be used for connecting to a list of distinct queue managers as that can result in transaction integrity issues.

Channel:

The name of the channel used for client connections to the WebSphere MQ queue manager, for client connection only.

Information	**Value**
Data type	String
Range	1 through 20 ASCII characters

Transport Type:

Whether WebSphere MQ client connection or JNDI bindings are used for connection to the WebSphere MQ queue manager.

Information	**Value**
Data type	Enum
Default	BINDINGS
Range	**CLIENT** WebSphere MQ client connection is used to connect to the WebSphere MQ queue manager. **BINDINGS** JNDI bindings are used to connect to the WebSphere MQ queue manager.

Client ID:

The JMS client identifier used for connections to the WebSphere MQ queue manager.

Information	**Value**
Data type	String

CCSID:

The coded character set identifier to use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs that WebSphere MQ supports. See the properties for the topic destination for more details.

Information	Value
Data type	String
Units	Integer
Range	1 through 65535

Broker Control Queue:

The name of the broker control queue to which all command messages (except publications and requests to delete publications) are sent.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker Queue Manager:

The name of the WebSphere MQ queue manager that provides the Publisher and Subscriber message broker.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker Publish Queue:

The name of the broker input queue that receives all publication messages for the default stream.

The name of the broker's input queue (stream queue) that receives all publication messages for the default stream. Applications can also send requests to delete publications on the default stream to this queue.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker Subscribe Queue:

The name of the broker queue from which nondurable subscription messages are retrieved.

The name of the broker queue from which nondurable subscription messages are retrieved. The subscriber specifies the name of the queue when it registers a subscription.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker CCSubQ:

The name of the broker queue from which nondurable subscription messages are retrieved for a ConnectionConsumer request. This property applies only for use of the web container.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Broker Version:

Whether the message broker is provided by the WebSphere MQ MA0C SupportPac or newer versions of WebSphere family message broker products.

Information	Value
Data type	Enum
Default	Advanced
Range	**Advanced** The message broker is provided by newer versions of WebSphere family message broker products (WebSphere MQ Integrator and WebSphere MQ Publish and Subscribe). **Basic** The message broker is provided by the WebSphere MQ MA0C SupportPac (WebSphere MQ - Publish and Subscribe).

Cleanup level:

The level of cleanup provided by the publish or subscribe cleanup utility.

Information	Value
Data type	Enum
Default	SAFE
Range	**ASPROP** **NONE** **STRONG**

Cleanup interval:

The interval, in milliseconds, between background executions of the publish/subscribe cleanup utility.

Information	Value
Data type	Integer
Units	Milliseconds
Default	6000

Message selection:

Where broker message selection is performed.

Information	Value
Data type	Enum
Default	BROKER

Information	Value
Range	**BROKER** Message selection is performed at the broker location. **Message CLIENT** Message selection is performed at the client location.

Publish acknowledge interval:

The interval, in number of messages, between publish requests that require acknowledgment from the broker.

Information	Value
Data type	Integer
Default	25

Sparse subscriptions:

Enables sparse subscriptions.

Information	Value
Data type	Check box
Default	Cleared

Status refresh interval:

The interval, in milliseconds, between transactions to refresh the publish or subscribe status.

Information	Value
Data type	Integer
Default	6000

Subscription store:

Where WebSphere MQ stores data relating to active JMS subscriptions.

Information	Value
Data type	Enum
Default	MIGRATE
Range	**MIGRATE** **QUEUE** **BROKER**

Multicast:

Whether this connection factory uses multicast transport.

Information	Value
Data type	Enum

Information	Value
Default	NOT USED
Range	**NOT USED** This connection factory does not use multicast transport. **ENABLED** This connection factory always uses multicast transport. **ENABLED_IF_AVAILABLE** This connection factory uses multicast transport. **ENABLED_RELIABLE** This connection factory uses reliable multicast transport. **ENABLED_RELIABLE_IF_AVAILABLE** This connection factory uses reliable multicast transport if available.

Direct authentication:

Whether to use direct broker authorization.

Information	Value
Data type	Enum
Default	NONE
Range	**NONE** Direct broker authorization is not used. **PASSWORD** Direct broker authorization is authenticated with a password. **CERTIFICATE** Direct broker authorization is authenticated with a certificate.

Proxy Host Name:

The host name of a proxy to be used for communication with WebSphere MQ.

Information	Value
Data type	String

Proxy Port:

The port number of a proxy to be used for communication with WebSphere MQ.

Information	Value
Data type	Integer
Default	0

Fail if quiesce:

Whether applications return from a method call if the queue manager has entered a controlled failure.

Information	Value
Data type	Check box
Default	Selected

Local Server Address:

The local server address.

Information	Value
Data type	String

Polling Interval:

The interval, in milliseconds, between scans of all receivers during asynchronous message delivery.

Information	Value
Data type	Integer
Units	Milliseconds
Default	5000

Rescan interval:

The interval in milliseconds between which a topic is scanned to look for messages that have been added to a topic out of order.

The rescan interval controls the scanning for messages that have been added to a topic out of order with respect to a WebSphere MQ browse cursor.

Information	Value
Data type	Integer
Units	Milliseconds
Default	5000

SSL cipher suite:

The cipher suite to use for SSL connection to WebSphere MQ.

Set this property to a valid cipher suite provided by your JSSE provider. The value must match the CipherSpec specified on the SVRCONN channel as the **Channel** property.

You must set this property, if you set the **SSL Peer Name** property.

SSL certificate store:

A list of zero or more Certificate Revocation List (CRL) servers that are used to check for SSL certificate revocation. If you specify a value for this property, you must use WebSphere MQ JVM at Java 2 version 1.4.

The value is a space-delimited list of entries of the form:

`ldap://`*`hostname`*`:[`*`port`*`]`

A single slash (/) follows this value. If *port* is omitted, the default LDAP port of 389 is assumed. At connect-time, the SSL certificate presented by the server is checked against the specified CRL servers.

For more information about CRL security, see the information about “Working with Certificate Revocation Lists” in the *Security* section of the WebSphere MQ information center.

SSL peer name:

For SSL, a *distinguished name* skeleton that must match the name provided by the WebSphere MQ queue manager. The distinguished name is used to check the identifying certificate presented by the server at connection time.

If this property is not set, such certificate checking is performed.

The SSL peer name property is ignored if **SSL Cipher Suite** property is not specified.

This property is a list of attribute name and value pairs separated by commas or semicolons. For example:

```
CN=QMGR.*, OU=IBM, OU=WEBSPHERE
```

The example given checks the identifying certificate presented by the server at connect-time. For the connection to succeed, the certificate must have a Common Name beginning QMGR., and must have at least two Organizational Unit names, the first of which is IBM and the second WEBSPHERE. Checking is not case-sensitive.

For more details about distinguished names and their use with WebSphere MQ, see the information about “Distinguished Names” in the Security section of the WebSphere MQ information center.

Connection pool:

An optional set of connection pool settings.

Connection pool properties are common to all J2C connectors.

The application server pools connections and sessions with the JMS provider to improve performance. This connection pooling is independent from any WebSphere MQ connection pooling. You must configure the connection and session pool properties appropriately for your applications, otherwise you might not get the connection and session behavior that you want.

Change the size of the connection pool if concurrent server-side access to the JMS resource exceeds the default value. The size of the connection pool is set on a per queue or topic basis.

Information	**Value**
Data type	Check box
Default	Selected

Client reconnect options:

Specifies whether a client mode connection reconnects automatically, or not, in the event of a communications or queue manager failure. This property is ignored unless the connection factory is being used in a thin or managed client environment.

Information	**Value**
Data type	Drop-down list
Default	DISABLED

Information	Value
Range	**DISABLED** The client reconnection does not automatically occur. **ASDEF** The value from the DefRecon attribute from the channels stanza of the client configuration file is used. If there is no DefRecon value specified then this setting has the same effect as a value of DISABLED. **RECONNECT** Reconnection occurs to any queue manager consistent with the value of the queue manager attribute, which might be a different queue manager from that to which the connection was originally connected. **QMGR** Reconnection only occurs to the queue manager to which the connection was originally connected.

For more information about automatic client reconnection, see the WebSphere MQ information center.

Client reconnect timeout:

The maximum number of seconds that a client mode connection spends attempting to automatically reconnect to a queue manager after a communications or queue manager failure. This parameter is ignored unless the connection factory is being used in a thin or managed client environment. Whether this parameter is used or not depends on the value of the client reconnect options parameter.

Information	Value
Data type	Integer
Units	Seconds
Default	1800
Range	A value greater than zero and up to 2147483647

For more information about automatic client reconnection, see the WebSphere MQ information center.

WebSphere MQ Provider queue destination settings for application clients:

Use this panel to view or change the configuration properties of the selected queue destination for use with the WebSphere MQ product Java Message Service (JMS) provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file and click **Messaging Providers** > **WebSphere MQ Provider**. Right-click **Queue Destinations** and click **New**. The following fields are displayed on the **General** tab.

Note:

- The property values that you specify must match the values that you specified when configuring JMS resources for WebSphere MQ. For more information about configuring JMS resources for WebSphere MQ, see *Using Java* in the WebSphere MQ information center.
- In WebSphere MQ, names can have a maximum of 48 characters.

A queue for use with the WebSphere MQ product JMS provider has the following properties.

Name:

The name by which the queue is known for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

Description:

A description of the queue, for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The application client runtime environment uses this field to retrieve configuration information.

Persistence:

Whether all messages sent to the destination are persistent, nonpersistent or have their persistence defined by the application.

Information	Value
Data type	Enum
Default	APPLICATION_DEFINED
Range	**Application defined** Messages on the destination have their persistence defined by the application that put them onto the queue. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Persistent** Messages on the destination are persistent. **Nonpersistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property.

Information	Value
Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property. *If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, specify the message priority for this queue, in the range 0 (lowest) through 9 (highest).

Information	Value
Data type	Integer
Units	Message priority level
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout value for this queue is defined by the application or the by **Specified expiry** property or whether messages on the queue never expire (have an unlimited expiry time out).

Information	Value
Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property. If you select this option, you must define a timeout on the **Specified expiry** property. **Unlimited** Messages on this queue have no expiry timeout and those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire.

Information	Value
Data type	Integer
Units	Milliseconds
Range	Greater than or equal to 0 • 0 indicates that messages never time out • Other values are an integer number of milliseconds

Base Queue Name:

The name of the queue to which messages are sent, on the queue manager specified by the **Base queue manager name** property.

Information	Value
Data type	String

Base Queue Manager Name:

The name of the WebSphere MQ queue manager to which messages are sent.

This queue manager provides the queue specified by the **Base queue name** property.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	A valid WebSphere MQ Queue Manager name, as 1 through 48 ASCII characters

CCSID:

The coded character set identifier to use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ queue manager. See the WebSphere MQ messaging provider queue and topic advanced properties settings for more details.

Information	Value
Data type	String

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Information	Value
Data type	Enum
Default	NORMAL
Range	**NORMAL** Normal integer encoding is used. **REVERSED** Reversed integer encoding is used. For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Information	Value
Data type	Enum
Default	NORMAL
Range	**NORMAL** Normal decimal encoding is used. **REVERSED** Reversed decimal encoding is used. For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Floating point encoding:

If native encoding is not enabled, select the type of floating point encoding.

Information	Value
Data type	Enum
Default	IEEENORMAL

Information	Value
Range	**IEEENORMAL** IEEE normal floating point encoding is used. **IEEEREVERSED** IEEE reversed floating point encoding is used. **S390** S390 floating point encoding is used. For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Native encoding:

Indicates that the queue destination uses native encoding (appropriate encoding values for the Java platform) when you select this check box.

Information	Value
Data type	Enum
Default	Cleared
Range	**Cleared** Native encoding is not used, so specify the following properties for integer, decimal and floating point encoding. **Selected** Native encoding is used (to provide appropriate encoding values for the Java platform). For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Target client:

Whether the receiving application is JMS compliant or is a traditional WebSphere MQ application.

Information	Value
Data type	Enum
Default	WebSphere MQ
Range	**WebSphere MQ** The target is a traditional WebSphere MQ application that does not support JMS. **JMS** The target application supports JMS.

Message body:

Specifies whether an application processes the RFH version 2 header of a WebSphere MQ message as part of the JMS message body.

Information	Value
Data type	Drop-down list
Default	UNSPECIFIED

Information	Value
Range	**UNSPECIFIED** When sending messages, the WebSphere MQ messaging provider does or does not generate and include an RFH version 2 header, depending on the value of the Append RFH version 2 headers to messages sent to this destination property. When receiving messages, the WebSphere MQ messaging provider acts as if the value is set to JMS. **JMS** When sending messages, the WebSphere MQ messaging provider automatically generates an RFH version 2 header and includes it in the WebSphere MQ message. When receiving messages, the WebSphere MQ messaging provider sets the JMS message properties according to values in the RFH version 2 header (if these value are present); it does not present the RFH version 2 header as part of the JMS message body. **MQ** When sending messages, the WebSphere MQ messaging provider does not generate an RFH version 2 header. When receiving messages, the WebSphere MQ messaging provider presents the RFH version 2 header as part of the JMS message body.

ReplyTo destination style:

Specifies the format of the JMSReplyTo field.

Information	Value
Data type	Drop-down list
Default	DEFAULT
Range	**DEFAULT** The default value is equivalent to the information in the RFH version 2 header. **MQMD** Use the value supplied in the MQMD. This populates the reply to queue manager field with the value from the MQMD, equivalent to the default behaviour of WebSphere MQ Version 6.0.2.4 and 6.0.2.5. **RFH2** Use the value supplied in the RFH version 2 header. If the sending application set a JMSReplyTo value, then that value is used.

MQMD read enabled:

Specifies whether an application can read the values of MQMD fields from JMS messages that have been sent or received using the WebSphere MQ messaging provider.

Information	Value
Data type	Check box
Default	Cleared
Range	**Cleared** Applications cannot read the values of the MQMD fields. **Selected** Applications can read the values of the MQMD fields.

MQMD write enabled:

Specifies whether an application can write the values of MQMD fields to JMS messages that will be sent or received using the WebSphere MQ messaging provider.

Information	Value
Data type	Check box

Information	Value
Default	Cleared
Range	**Cleared** Applications cannot write the values of the MQMD fields. **Selected** Applications can write the values of the MQMD fields.

MQMD message context:

Defines the message context options specified when sending messages to a destination.

Information	Value
Data type	Drop-down list
Default	DEFAULT
Range	**DEFAULT** The MQOPEN API call and the MQPMO structure specify no explicit message context options. **SET_IDENTITY_CONTEXT** The MQOPEN API call specifies the message context option MQOO_SET_IDENTITY_CONTEXT, and the MQPMO structure specifies MQPMO_SET_IDENTITY_CONTEXT. **SET_ALL_CONTEXT** The MQOPEN API call specifies the message context option MQOO_SET_ALL_CONTEXT, and the MQPMO structure specifies MQPMO_SET_ALL_CONTEXT.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere MQ Provider topic destination settings for application clients:

Use this panel to view or change the configuration properties of the selected topic destination for use with the WebSphere MQ product Java Message Service (JMS) provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > **WebSphere MQ Provider**. Right click **Topic Destinations**, and click **New**. The following fields are displayed on the **General** tab.

Note:

- The property values that you specify must match the values that you specified when configuring JMS resources for WebSphere MQ. For more information about configuring JMS resources for WebSphere MQ, see *Using Java* in the WebSphere MQ information center.
- In WebSphere MQ, names can have a maximum of 48 characters.

A topic destination is used to configure the properties of a JMS topic for the associated JMS provider. A topic for use with the WebSphere MQ product JMS provider has the following properties.

Name:

The name by which the topic is known for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

Description:

A description of the topic for administrative purposes within WebSphere Application Server.

Information	Value
Data type	String

JNDI Name:

The application client runtime environment uses this field to retrieve configuration information.

Persistence:

Whether all messages sent to the destination are persistent, nonpersistent, or have their persistence defined by the application.

Information	Value
Data type	Enum
Default	APPLICATION_DEFINED
Range	**Application defined** Messages on the destination have their persistence defined by the application that put them in the queue. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Persistent** Messages on the destination are persistent. **Nonpersistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property.

Information	Value
Data type	Enum
Default	APPLICATION_DEFINED
Range	**Application defined** The priority of messages on this destination is defined by the application that put them in the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property. If you select this option, you must define a priority for the **Specified priority** property.

Specified Priority:

If the **Priority** property is set to `Specified`, specify the message priority for this queue, in the range 0 (lowest) through 9 (highest).

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Information	Value
Data type	Integer
Units	Message priority level
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or by the **Specified expiry** property, or whether messages on the queue never expire (have an unlimited expiry timeout).

Information	Value
Data type	Enum
Default	APPLICATION_DEFINED
Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them in the queue. **Specified** The expiry timeout for messages in this queue is defined by the **Specified expiry** property. If you select this option, you must define a timeout value for the **Specified expiry** property. **Unlimited** Messages on this queue have no expiry timeout, and these messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type the number of milliseconds (greater than 0) after which messages on this queue expire.

Information	Value
Data type	Integer
Units	Milliseconds
Range	Greater than or equal to 0 • 0 indicates that messages never time out. • Other values are an integer number of milliseconds.

Base Topic Name:

The name of the topic to which messages are sent.

Information	Value
Data type	String

CCSID:

The coded character set identifier to use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs that WebSphere MQ supports.

Information	Value
Data type	String
Units	Integer
Range	1 through 65535

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Information	Value
Data type	Enum
Default	NORMAL
Range	**NORMAL** Normal integer encoding is used. **REVERSED** Reversed integer encoding is used. For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Information	Value
Data type	Enum
Default	NORMAL
Range	**NORMAL** Normal decimal encoding is used. **REVERSED** Reversed decimal encoding is used. For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Floating point encoding:

If native encoding is not enabled, select the type of floating point encoding.

Information	Value
Data type	Enum
Default	IEEENORMAL
Range	**IEEENORMAL** IEEE normal floating point encoding is used. **IEEEREVERSED** IEEE reversed floating point encoding is used. **S390** S/390® floating point encoding is used. For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

Native encoding:

Indicates that the queue destination uses native encoding (appropriate encoding values for the Java platform) when you select this check box.

Information	Value
Data type	Enum
Default	Cleared
Range	**Cleared** Native encoding is not used, so specify the previous properties for integer, decimal and floating point encoding. **Selected** Native encoding is used (to provide appropriate encoding values for the Java platform). For more information about encoding properties, see *Using Java* in the WebSphere MQ information center.

BrokerDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved.

The subscriber specifies the name of the queue when it registers a subscription.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

BrokerCCDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the web container.

Information	Value
Data type	String
Units	En_US ASCII characters
Range	1 through 48 ASCII characters

Target Client:

Whether the receiving application is JMS compliant or is a traditional WebSphere MQ application.

Information	Value
Data type	Enum
Default	WebSphere MQ
Range	**WebSphere MQ** The target is a traditional WebSphere MQ application that does not support JMS. **JMS** The target application supports JMS.

Message body:

Specifies whether an application processes the RFH version 2 header of a WebSphere MQ message as part of the JMS message body.

Information	Value
Data type	Drop-down list
Default	UNSPECIFIED

Information	Value
Range	**UNSPECIFIED** When sending messages, the WebSphere MQ messaging provider does or does not generate and include an RFH version 2 header, depending on the value of the Append RFH version 2 headers to messages sent to this destination property. When receiving messages, the WebSphere MQ messaging provider acts as if the value is set to JMS. **JMS** When sending messages, the WebSphere MQ messaging provider automatically generates an RFH version 2 header and includes it in the WebSphere MQ message. When receiving messages, the WebSphere MQ messaging provider sets the JMS message properties according to values in the RFH version 2 header (if these value are present); it does not present the RFH version 2 header as part of the JMS message body. **MQ** When sending messages, the WebSphere MQ messaging provider does not generate an RFH version 2 header. When receiving messages, the WebSphere MQ messaging provider presents the RFH version 2 header as part of the JMS message body.

ReplyTo destination style:

Specifies the format of the JMSReplyTo field.

Information	Value
Data type	Drop-down list
Default	DEFAULT
Range	**DEFAULT** The default value is equivalent to the information in the RFH version 2 header. **MQMD** Use the value supplied in the MQMD. This populates the reply to queue manager field with the value from the MQMD, equivalent to the default behaviour of WebSphere MQ Version 6.0.2.4 and 6.0.2.5. **RFH2** Use the value supplied in the RFH version 2 header. If the sending application set a JMSReplyTo value, then that value is used.

Multicast:

Whether this connection factory uses multicast transport.

Information	Value
Data type	Enum
Default	AS_CF
Range	**AS_CF** This connection factory uses multicast transport. **DISABLED** This connection factory does not use multicast transport. **NOT_RELIABLE** This connection factory always uses multicast transport. **RELIABLE** This connection factory uses multicast transport when the topic destination is not reliable. **ENABLED** This connection factory uses reliable multicast transport.

MQMD read enabled:

Specifies whether an application can read the values of MQMD fields from JMS messages that have been sent or received using the WebSphere MQ messaging provider.

Information	Value
Data type	Check box
Default	Cleared
Range	**Cleared** Applications cannot read the values of the MQMD fields. **Selected** Applications can read the values of the MQMD fields.

MQMD write enabled:

Specifies whether an application can write the values of MQMD fields to JMS messages that will be sent or received using the WebSphere MQ messaging provider.

Information	Value
Data type	Check box
Default	Cleared
Range	**Cleared** Applications cannot write the values of the MQMD fields. **Selected** Applications can write the values of the MQMD fields.

MQMD message context:

Defines the message context options specified when sending messages to a destination.

Information	Value
Data type	Drop-down list
Default	DEFAULT
Range	**DEFAULT** The MQOPEN API call and the MQPMO structure specify no explicit message context options. **SET_IDENTITY_CONTEXT** The MQOPEN API call specifies the message context option MQOO_SET_IDENTITY_CONTEXT, and the MQPMO structure specifies MQPMO_SET_IDENTITY_CONTEXT. **SET_ALL_CONTEXT** The MQOPEN API call specifies the message context option MQOO_SET_ALL_CONTEXT, and the MQPMO structure specifies MQPMO_SET_ALL_CONTEXT.

Generic JMS connection factory settings for application clients:

Use this panel to view or change the configuration properties of the selected Java Message Service (JMS) connection factory for use with the associated JMS provider. These configuration properties control how connections are created between the JMS provider and the messaging system that it uses.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > *new_JMS_Provider_instance*. Right-click **Connection Factories**, and click **New**. The following fields are displayed on the **General** tab.

A Java Message Service (JMS) connection factory creates connections to JMS destinations. The JMS connection factory is created by the associated JMS provider. A JMS connection factory for a generic JMS provider (other than the internal default messaging provider or WebSphere MQ as a JMS provider) has the following properties:

Name:

The name by which this JMS connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the associated JMS provider.

Information	Value
Data type	String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Information	Value
Data type	String
Default	Null

JNDI Name:

The application client run time uses this field to retrieve configuration information.

User ID:

Indicates the user ID used with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a userid and password explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Information	Value
Data type	String

Password:

The password used with the **User ID** property for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Information	Value
Data type	String
Default	Null

Re-Enter Password:

Confirms the password entered in the **Password** field.

External JNDI Name:

The JNDI name that is used to bind the queue into the application server name space.

As a convention, use the fully qualified JNDI name, for example, `jms/`*`Name`*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI API by the platform.

Information	**Value**
Data type	String

Connection Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for publication or subscription).

Select one of the following options:

Queue
A JMS queue destination for point-to-point messaging.

Topic A JMS topic destination for publish subscribe messaging.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Generic JMS destination settings for application clients:

Use this panel to view or change the configuration properties of the selected JMS destination for use with the associated JMS provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Messaging Providers** > *new JMS Provider instance*. Right-click **Destinations**, and click **New**. The following fields are displayed on the **General** tab.

A JMS destination is used to configure the properties of a JMS destination for the associated generic JMS provider. Connections to the JMS destination are created by the associated JMS connection factory. A JMS destination for use with a generic JMS provider (not the default messaging provider or WebSphere MQ as a JMS provider) has the following properties.

Name:

The name by which the queue is known for administrative purposes within WebSphere Application Server.

Information	**Value**
Data type	String

Description:

A description of the queue, for administrative purposes.

JNDI Name:

The JNDI name of the actual (physical) name of the JMS destination bound into JNDI.

External JNDI Name:

The JNDI name that is used to bind the queue into the application server name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Information	Value
Data type	String

Destination Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for publishing or subscribing).

Select one of the following options:
Queue
A JMS queue destination for point-to-point messaging.
Topic A JMS topic destination for pub/sub messaging.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring JMS provider, JMS connection factory and JMS destination settings for application clients:

You can configure JMS Provider, JMS Connection Factory and JMS Destination settings. This topic provides the required fields, special cases, and an example.

The purpose of this article is to help you to configure JMS Provider, JMS Connection Factory and JMS Destination settings.
- Required fields include:
  - JMS Provider Properties page: name, and at least one protocol provider
  - JMS Connection Factory Properties page: name, jndiName, destination type
  - JMS Destination Properties page: name, jndiName, destination type
- Special cases:
  - The destination type must be `QUEUE`, or `TOPIC`.
- Example:

```
<resources.jms:JMSProvider xmi:id="JMSProvider_3" name="genericJMSProvider:name"
description="genericJMSProvider:description"
externalInitialContextFactory="genericJMSProvider:contextFactoryClass"
```

```
externalProviderURL="genericJMSProvider:providerUrl">
<classpath>genericJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms:GenericJMSDestination"
xmi:id="GenericJMSDestination_1" name="jmsDestination:name"
jndiName="jmsDestination:jndiName" description="jmsDestination:description"
externalJNDIName="jmsDestination:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_15">
<resourceProperties xmi:id="J2EEResourceProperty_17" name="jmsDestination:customName"
value="jmsDestination:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms:GenericJMSConnectionFactory"
xmi:id="GenericJMSConnectionFactory_1" name="jmsCF:name" jndiName="jmsCF:jndiName"
description="jmsCF:description" userID="jmsCF:user" password="{xor}NTIsHBllMT4yOg=="
externalJNDIName="jmsCF:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_16">
<resourceProperties xmi:id="J2EEResourceProperty_18" name="jmsCF:customName"
value="jmsCF:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_17">
<resourceProperties xmi:id="J2EEResourceProperty_19"
name="genericJMSProvider:customName" value="genericJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Configuring new JMS connection factories for application clients

Use this task to create a new Java Message Service (JMS) connection factory configuration for your application client.

Procedure

1. Click the JMS provider for which you want to create a connection factory in the tree. Complete one of the following actions:
   - Configure a new JMS provider.
   - Click an existing JMS provider.
2. Expand the JMS provider to view its Connection Factories folder.
3. Click the connection factory folder, and complete one of the following actions:
   - Right-click the folder and select**New**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS connection factory properties in the displayed fields.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring new JMS destinations for application clients

Use this task to create a new Java Message Service (JMS) destination configuration for your application client.

Procedure

1. Click the JMS provider in the tree for which you want to create a destination. Complete one of the following actions:
   - Configure a new JMS provider.
   - Click an existing JMS provider.
2. Expand the JMS provider to view its Destinations folder.
3. Click the provider folder, and complete one of the following actions:
   - Right-click the folder and select **New**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS destination properties in the displayed fields.

5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring new resource environment providers for application clients

You can create new resource environment provider configurations for your application client using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create new resource environment provider configurations for your application client.

About this task

To configure a new resource environment provider, perform the following steps:

Procedure

1. Start the Application Configuration Resource Tool and open the EAR file for which you want to configure the new Java Message Service (JMS) provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file in which you want to configure the new JMS provider.
3. Expand the JAR file to view its contents.
4. Click the **Resource Environment Providers** folder. Take one of the following actions:
   - Right-click the provider folder, and click **New**.
   - Click **Edit > New** on the menu bar.
5. Configure the JMS provider properties in the displayed fields.
6. Click **OK** when you finish.
7. Click **File > Save** on the menu bar to save your changes.

Resource environment provider settings for application clients:

Use this page to specify resource environment entry properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected Java Archive (JAR) file. Right-click **Resource Environment Providers**, and click **New**. The following fields are displayed on the **General** tab:

Name:

Specifies the administrative name for the resource environment provider.

Description:

Specifies a description of the resource environment provider for your administrative records.

Class Path:

Specifies the path to the JAR file that contains the implementation classes for the resource environment provider.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new resource environment entries for application clients

You can create new resource environment entries for your client application using the Application Client Resource Configuration Tool (ACRCT).

Before you begin

During this task, you create new resource environment entries for your client application.

About this task

Procedure

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new resource environment entry. The EAR file contents are in the displayed tree view.
3. Click the desired resource environment provider, and complete the following action to configure new providers:
   - Configure a new resource environment provider.
4. Expand the resource environment provider to view the Resource Environment Entries folder.
5. Click the resource environment entries folder, and complete one of the following actions:
   - Right-click the folder and select **New**.
   - Click **Edit > New** on the menu bar.
6. Configure the resource environment entry properties in the displayed fields.
7. Click **OK**.
8. Click **File > Save** on the menu bar to save your changes.

Resource environment entry settings for application clients:

Use this page to specify resource environment entry properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Environment Providers** > *resource environment instance*. Right-click **Resource Environment Entries**, and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the administrative name for the resource environment entry.

Description:

Specifies a description of the URL for your administrative records.

JNDI Name:

Specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts.

Use this name to link to the binding information of the platform. The binding associates the resources defined in the deployment descriptor of the module to the actual (or physical) resources bound into JNDI by the platform.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring Resource Environment settings:

You can configure Resource Environment settings. This topic provides the required fields and an example.

The purpose of this topic is to help you configure Resource Environment settings.
- Required fields:
  - Resource Environment Provider page: **Name**
  - Resource Environment Entry page: **Name, JNDI Name**
- Example:

```
<resources.env:ResourceEnvironmentProvider xmi:id="ResourceEnvironmentProvider_1"
name="resourceEnvProvider:name" description="resourceEnvProvider:description">
<classpath>resourceEnvProvider:classpath</classpath>
<factories xmi:type="resources.env:ResourceEnvEntry" xmi:id="ResourceEnvEntry_1"
name="resourceEnvEntry:name" jndiName="resourceEnvEntry:jndiName"
description="resourceEnvEntry:description">
<propertySet xmi:id="J2EEResourcePropertySet_20">
<resourceProperties xmi:id="J2EEResourceProperty_22"
name="resourceEnvEntry:customName" value="resourceEnvEntry:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_21">
<resourceProperties xmi:id="J2EEResourceProperty_23"
name="resourceEnvProvider:customName" value="resourceEnvProvider:customValue"/>
</propertySet>
</resources.env:ResourceEnvironmentProvider>
```

Example: Configuring resource environment custom settings for application clients:

You can configure resource environment custom settings.

The purpose of this topic is to help you configure resource environment custom settings.
- The custom page applies to every resource type. You can specify as many custom names and values as you need.
- Example:

```
<propertySet xmi:id="J2EEResourcePropertySet_20">
<resourceProperties xmi:id="J2EEResourceProperty_22"
name="resourceEnvEntry:customName" value="resourceEnvEntry:customValue"/>
</propertySet>
```

Running a Java EE client application with launchClient

After deploying a Java EE client application onto a machine with an Application Client installation or in a WebSphere Application Server node, you can start the application by using the **launchClient** command on that machine.

Before you begin

Before you can use the **launchclient** command to run a Java EE client application, you must have deployed the application.

This task only applies to Java EE client applications.

About this task

The Java Platform, Enterprise Edition (Java EE) specification requires support for a client container that runs Java applications (known as Java EE client applications) and provides Java EE services to the applications. Java EE services include naming, security, and resource connections.

Procedure

1. Enter the following command to launch Java EE application clients:

   `app_client_root/bin/launchClient`

2. Pass parameters to the `launchClient` command or to your application client program as well. The `launchClient` command allows you to do both. The `launchClient` command requires that the first parameter is either:
   - An EAR file specifying the application client to launch.
   - A request for `launchClient` usage information.

   The following example illustrates the command line invocation syntax for the launchClient tool:

   `launchClient [-profileName pName | -JVMOptions options | -help | -?] userapp [-CCname=value] [app args]`

   where

   - *userapp* is the path and the name of the EAR file that contains the application client.
   - *-CCname=value* is the client container name-value pair parameter. See the client container parameters section, for supported name-value pair arguments.
   - *app args* are arguments that pass to the application client.
   - *-profileName* defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment or in an Application Clients installation.

     The default is default_profile.
   - *-JVMOptions* is a valid Java standard or non-standard option string. Insert quotation marks around the string.
   - *-help, -?* prints the usage information.

   All other parameters intended for the `launchClient` command must begin with the -CC prefix.

   Parameters that are not EAR files, or usage requests, or that do not begin with the -CC prefix, are ignored by the application client run time, and are passed directly to the application client program.

   The `launchClient` command retrieves parameters from three places:
   - The command line
   - A properties file
   - System properties

   The parameters are resolved in the order listed above, with command line values having the highest priority and system properties the lowest. Using this prioritization you can set and override default values.

3. Specify the server name.

   By default, the launchClient command uses the localhost for the `BootstrapHost` property value.

   This setting is effective for testing your application client when it is installed on the same computer as the server. However, in other cases override this value with the name of your server. You can override the `BootstrapHost` value by invoking `launchClient` command with the following parameters:

   ```
   launchClient myapp.ear -CCBootstrapHost=abc.midwest.mycompany.com
   ```

   You can also override the default by specifying the value in a properties file and passing the file name to the launchClient shell.

Security is controlled by the server. You do not need to configure security on the client because the client assumes that security is enabled. If server security is not enabled, then the server ignores the security request, and the application client functions as expected.

Example

You can store launchClient values in a properties file, which is a good method for distributing default values. You can then override one or more values on the command line. The format of the file is one `launchClient -CC` parameter per line without the `-CC` prefix. For example:

```
 verbose=true classpath=/usr/lpp/mydir/util.jar;/usr/lpp/mydir/harness.jar;/usr/lpp
/production/G19/global.jar BootstrapHost=abc.westcoast.mycompany.com tracefile=/usr
/lpp/WebSphere/mylog.txt
```

launchClient tool

This topic describes the Java Platform, Enterprise Edition (Java EE) command line syntax for the launchClient tool for WebSphere Application Server.

You can use the **launchClient** command from a node within a WebSphere Application Server, Network Deployment environment. However, do not attempt to use the **launchClient** command from the Deployment Manager.

Important: All users who run commands from a specific profile must have authority to modify files that are created by other users that use the same profile. Otherwise, you might see a permission denied error in the log files. To avoid this issue, consider one of the following policies:

- Use a separate installation for distinct user authorities
- Always use the same user for all of the commands that are run in a given profile
- Ensure that all users of a specific profile belong to the same group. In addition, ensure that each user of a group has the read and write authority to the files that are created by other members in the same profile.

The following example illustrates the command line invocation syntax for the launchClient tool:

`launchClient [-profileName pName | -JVMOptions options | -help | -?] userapp [-CCname=value] [app args]`

where

- *userapp* is the path and the name of the EAR file that contains the application client.
- *-CCname=value* is the client container name-value pair parameter. See the client container parameters section, for supported name-value pair arguments.
- *app args* are arguments that pass to the application client.
- *-profileName* defines the profile of the Application Server process in a multi-profile installation. The -profileName option is not required for running in a single profile environment or in an Application Clients installation.

  The default is `default_profile`.
- *-JVMOptions* is a valid Java standard or nonstandard option string, except `-cp` or `-classpath`. Insert quotation marks around the string.
- *-help, -?* prints the usage information.

The first parameter must be `-help`, `-?` or contain no parameter at all. The `-profileName pName` and `-JVMOptions` options are optional parameters. If used, they must appear before the `<userapp>` parameter. All other parameters are optional and can appear in any order after the *userapp* parameter. The Java EE Application client run time ignores any optional parameters that do not begin with a `-CC` prefix and passes those parameters to the application client.

Client container parameters

Supported arguments include:

-CCadminConnectorHost
Specifies the host name of the server from which configuration information is retrieved.

The default is the value of the -CCBootstrapHost parameter or the value, localhost, if the -CCBootstrapHost parameter is not specified.

-CCadminConnectorPort
Indicates the port number for the administrative client function to use. The default value is 8880 for SOAP connections and 2809 for Remote Method Invocation (RMI) connections.

-CCadminConnectorType
Specifies how the administrative client connects to the server. Specify RMI to use the RMI connection type, or specify SOAP to use the SOAP connection type. The default value is SOAP.

-CCadminConnectorUser
Administrative clients use this user name when a server requires authentication. If the connection type is SOAP, and security is enabled on the server, this parameter is required.

-CCadminConnectorPassword
The password for the user name that the -CCadminConnectorUser parameter specifies.

-CCaltDD
The name of an alternate deployment descriptor file. This parameter is used with the -CCjar parameter to specify the deployment descriptor to use. Use this argument when a client JAR file is configured with more than one deployment descriptor. Set the value to null to use the client JAR file standard deployment descriptor.

-CCBootstrapHost
The name of the host server you want to connect to initially. The format is:
your_server_of_choice.com

-CCBootstrapPort
The server port number. If you do not specify this argument, the WebSphere Application Server default value is used.

-CCclassLoaderMode
Specifies the class loader mode. If PARENT_LAST is specified, the class loader loads classes from the local class path before delegating the class loading to its parent. The classes loaded for the following are affected:

- Classes defined for the Java EE application client
- Resources defined in the Java EE application
- Classes specified on the manifest of the Java EE client JAR file
- Classes specified using the -CCclasspath option

If PARENT_LAST is not specified, then the default mode, PARENT_FIRST, causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path.

-CCclasspath
A class path value. When you launch an application, the system class path is used. If you want to access classes that are not in the EAR file or part of the system class paths, specify the appropriate class path here. Multiple paths can be concatenated.

-CCD
Use this option to have the WebSphere Application Server set the specified system property during initialization. Do not use the equals (=) character after the -CCD. For example:

-CCDcom.ibm.test.property=testvalue. You can specify multiple -CCD parameters. The general format of this parameter is -CCD<property key>=<property value>. For example, -CCDI18NService.enable=true.

-CCdumpJavaNameSpace
Controls generation of a dump of the java: name space for the application that is launched, which can be used for debugging purposes. A value of **true** generates a dump in short format, and includes the name and object type for each binding. A value of **long** generates a dump in long format, and includes additional information for each binding over short format, such as the local object type and string representation of the local object. The default value is **false**, and does not generate a dump.

-CCexitVM
Use this option to have the WebSphere Application Server call the System.exit() method after the client application completes. The default is false.

-CCinitonly
Use this option to initialize application client run time for ActiveX application clients without launching the client application. The default is false.

-CCjar
The name of the client Java Archive (JAR) file that resides within the EAR file for the application you wish to launch. Use this argument when you have multiple client JAR files in the EAR file.

-CCpropfile
Indicates the name of a properties file that contains launchClient properties. Specify the properties without the -CC prefix in the file, with the exception of the securityManager, securityMgrClass and securityMgrPolicy properties. See the following example: verbose=true.

-CCproviderURL
Provides bootstrap server information that the initial context factory can use to obtain an initial context. WebSphere Application Server initial context factory can use either a Common Object Request Broker Architecture (CORBA) object URL or an Internet Inter-ORB Protocol (IIOP) URL. CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. This value can contain more than one bootstrap server address. This feature can be used when attempting to obtain an initial context from a server cluster. You can specify bootstrap server addresses, for all servers in the cluster, in the URL. The operation will succeed if at least one of the servers is running, eliminating a single point of failure. The address list does not process in a particular order. For naming operations, this value overrides the -CCBootstrapHost and -CCBootstrapPort parameters. A CORBA object URL specifying multiple systems is illustrated in the following example:

```
-CCproviderURL=corbaloc:iiop:myserver.mycompany.com:9810,:mybackupserver.mycompany.com:2809
```

This value is mapped to the java.naming.provider.url system property.

-CCsecurityManager
Enables and runs the WebSphere Application Server with a security manager. The default is disable.

-CCsecurityMgrClass
Indicates the fully qualified name of a class that implements a security manager. Only use this argument if the -CCsecurityManager parameter is set to enable. The default is java.lang.SecurityManager.

-CCsecurityMgrPolicy
Indicates the name of a security manager policy file. Only use this argument if the -CCsecurityManager parameter is set to enable. When you enable this parameter, the java.security.policy system property is set. The default is *app_server_root*/properties/client.policy.

-CCsoapConnectorPort
The Simple Object Access Protocol (SOAP) connector port. If you do not specify this argument, the WebSphere Application Server default value is used.

-CCtrace
Use this option to obtain debug trace information. You might need this information when reporting a problem to IBM customer support. The default is `false`. For more information, read the Enabling trace topic.

-CCtracefile
Indicates the name of the file to which trace information is written. The default is to write output to the console.

-CCtraceMode
Specifies the trace format to use for tracing. If the valid value, `basic`, is not specified the default is advanced. Basic tracing format is a more compact form of tracing.

-CCverbose
This option displays additional information messages. The default is `false`.

If you are using an EJB client application with security enabled, edit the `sas.client.props` file, which is located in the *profile_root*`/properties` directory. Within the file, change the `com.ibm.CORBA.loginSource` value to *none*.

For more information on the `sas.client.props` utility, refer to the Manually encoding passwords in properties files and the PropFilePasswordEncoder command reference topics.

RMI connection with security. Used with the EJB and administrative client application.

Using Jacl:

```
wsadmin.sh -conntype RMI -port  rmiportnumber -user userid
-password password
```

Using Jython:

```
wsadmin.sh -lang jython -conntype RMI -port  rmiportnumber -user userid
-password password
```

rmiportnumber for your connection displays in the administrative console as BOOTSTRAP_ADDRESS.

Attention: On the AIX, HP-UX, Linux, IBM i, Solaris, and z/OS operating systems, the use of -password option may result in security exposure as the password information becomes visible to the system status program, such as ps command, which can be invoked by other users to display all of the running processes. Do not use this option if security exposure is a concern. Instead, specify user and password information in the `soap.client.props` file for SOAP connector or `sas.client.props` file for RMI connector. The `soap.client.props` and `sas.client.props` files are located in the properties directory of your WebSphere Application Server profile.

If Kerberos (KRB5) is enabled for administrative authentication, the authentication target supports BasicAuth and KRB5. To use KRB5, update the `sas.client.props`, `soap.client.props`, and `ipc.client.props` files, according to the connector type.

Attention: When using Kerberos authentication, the user password does not flow across the wire. A one-way hash of password is used to identify the client.

The following examples demonstrate correct syntax.

```
./launchClient.sh /usr/earfiles/myapp.ear -CCBootstrapHost=myWASServer -CCverbose=true app_parm1 app_parm2
```

Specifying the directory for an expanded EAR file

You can archive the `Manifest.mf` client Java Archive (JAR) files instead of automatically cleaning them up after the application exits.

Before you begin

Each time the launchClient tool is called, it extracts the Enterprise Archive (EAR) file to a random directory name in the temporary directory on your hard drive. Then the tool sets up the thread ClassLoader to use the extracted EAR file directory and JAR files included in the `Manifest.mf` client Java Archive (JAR) file. In a normal J2EE Java client, these files are automatically cleaned up after the application exits. This cleanup occurs when the client container shutdown hook is called. To avoid extracting the EAR file (and removing the temporary directory) each time the launchClient tool is called, complete the following steps:

Procedure

1. Specify a directory to extract the EAR file by setting the `com.ibm.websphere.client.applicationclient.archivedir` Java system property. If the directory does not exist or is empty, the EAR file is extracted normally. If the EAR file was previously extracted, the launchClient tool reuses the directory.
2. Delete the directory before running the launchClient tool again, if you need to update your EAR file. When you call the `launchClient` command, it extracts the new EAR file to the directory. If you do not delete the directory or change the system property value to point to a different directory, the launchClient tool reuses the currently extracted EAR file and does not use your changed EAR file. When specifying the `com.ibm.websphere.client.applicationclient.archivedir` property, make sure that the directory you specify is unique for each EAR file you use. For example, do not point the `MyEar1.ear` and the `MyEar2.ear` files to the same directory.

Downloading and running a Java EE client application using Java Web Start

Learn about the Java Web Start technology that is provided by the Java Standard Edition runtime environment to deploy Java Enterprise Edition application clients, including Thin application clients, on the remote client machine with a single click from a web browser on the client machine.

Before you begin

The supported client platforms for deploying application clients using the Java Web Start are the same as the IBM Application Client for WebSphere Application Server supported platforms, except Linux on Power® and OS/400® operating systems.

Before you begin this task, see the following topics to understand Java Web Start technology and its components:

- “Java Web Start architecture for deploying application clients” on page 2023
- “Client application Java Network Launcher Protocol deployment descriptor file” on page 2024
- “ClientLauncher class” on page 2027

Note: The Sun Java Web Start, which is available from Sun Microsystems, is not compatible with the IBM Runtime Environment, Java 2 Technology Edition, which is provided by WebSphere Application Server and the IBM Application Client. The IBM Runtime Environment contains some additional functionality that is not supported in the Sun Java Web Start. Also, the IBM Runtime Environment uses a different packaging structure than the Sun Java Web Start. Use the IBM Runtime Environment.

About this task

To deploy application clients using Java Web Start, the client machine must have at least a Java SE runtime environment installed. The Java SE runtime environment includes the Java Web Start, which implements the JSR 56: Java Network Launching Protocol and API. The application clients Enterprise

Archive (EAR) file is a Java archive (JAR) resource in a JNLP descriptor file that resides on a central server. The JNLP descriptor file also specifies the runtime environment requirement for running the application.

WebSphere Application Server provides a launcher class to launch the Java EE application client in the application client container inside of Java Web Start. The client machine might not have the IBM Application Client for WebSphere Application Server installed. If this is the case, create and install an application client container and runtime package as a runtime environment through Java Web Start. The JNLP descriptor file specifies this runtime environment as the required runtime environment for running the Java EE application client.

WebSphere Application Server also provides command-line utility programs to create this application client container and runtime package from an existing IBM Application Client for WebSphere Application Server installation, as well as an installer class to install this package as a runtime environment for the application client container and also the Java Runtime Environment (JRE) in the IBM Application Client for WebSphere Application Server installation. To run the Java EE application client, the EAR file is deployed as a JAR resource that is described in the JNLP descriptor file.

Procedure

1. Identify the client machine operating system, and install the corresponding IBM Application Client for WebSphere Application Server on a development machine. For example, if the Java EE application clients are targeted to run on Windows operating systems, install the IBM Application Client for WebSphere Application Server for Windows.
2. Run the utility programs to create the application client container and runtime package.
   a. Use the “buildClientRuntime tool” on page 2032 utility to create the package.
   b. Use the “buildClientLibJars tool” on page 2024 utility to create the JAR files containing the launcher and the installer class. This utility also zips up the properties files in the <app_client_root>/ properties directory.
3. Create the runtime installer JNLP descriptor file. The JNLP response must be included in the JNLP version ID to indicate the current runtime version in the response header, for example, `x-java-jnlp-version-id=1.6.0`. Using a servlet of a JavaServer Pages (JSP) file to provide a dynamic JNLP response.
4. Create the Java EE application client launch JNLP descriptor file.
5. Package the application client container runtime environments and the Java EE application in an Enterprise Archive (EAR) file. Depending on your preferred deployment strategy, the files can be in two separate Web modules, or combined into one.
6. All JAR resources must be Java signed, including the Java EE application client EAR file.
7. Deploy the Enterprise Archive file on an application server, and start the application. The Java EE application client is ready to be deployed.

Example

A Java Web Start deployment Sample is included in the client samples. This Sample demonstrates the steps to deploy a Java EE application client with an automated ANT script. The Sample has a servlet to generate the runtime installer JNLP response with JNLP version ID, for example, `x-java-jnlp-version-id`.

Important: When the application client initially launches using Java Web Start from Sun Microsystems Java SE Runtime Environment 6.0, it installs the Application Client runtime, which includes the IBM JRE. An null pointer exception (NPE) is thrown from the `com.sun.deploy.services.WPlatformService.getSecureRandom()` method. This is a known bug in Sun Java SE 6 (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6505528). If you experience this exception, relaunch the application. The NPE only occurs on the first launch of the application client.

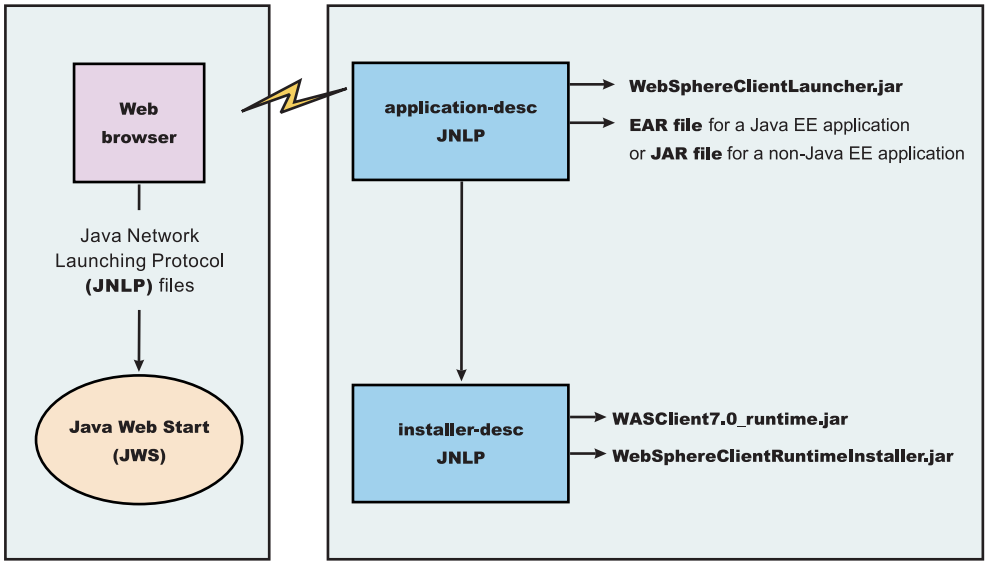
Java Web Start architecture for deploying application clients

Java Web Start is an application-deployment technology that includes the portability of applets, the maintainability of servlets and JavaServer Pages (JSP) file technology, and the simplicity of mark-up languages such as XML and HTML. It is a Java application that allows full-featured Java EE client applications to be launched, deployed and updated from a standard Web server. The Java Web Start client is used with platforms that support a web browser.

Upon launching Java Web Start for the first time, you might download new client applications from the Web. Each time you launch JWS thereafter, you can initiate applications either through a link on a web page or (in Windows) from desktop icons or the Start menu. You can deploy applications quickly using Java Web Start, cache applications on the client machine, and launch applications remotely offline. Additionally, because Java Web Start is built from the Java Platform, Enterprise Edition (Java EE) infrastructure, the technology inherits the complete security architecture of the Java EE platform.

The technology underlying Java Web Start is the Java Network Launching Protocol & API (JNLP). Java Web Start is a JNLP client and it reads and parses a JNLP descriptor file (JNLP file). Based on the JNLP descriptor, it downloads appropriate pieces of a client application and any of its dependencies. If any of the pieces of the application are already cached on the client machine, then those components are not downloaded again, unless they have been updated on the server machine. After you download and cache the client application, JWS launches it natively on the client machine.

The following diagram shows an overview of launching a client application, include the Application Client for WebSphere Application Server as a dependent resource, using Java Web Start.



The web browser running on a client machine connects to a web application located on a server machine. The client application JNLP descriptor file is downloaded and processed by Java Web Start on the client machine.

In this diagram, there are two JNLP descriptor files:

- Client application JNLP descriptor (application-desc in the diagram)
- Application Clients run-time installer JNLP descriptor (installer-desc in the diagram)

Each of these JNLP descriptor files, the client application (JAR or EAR) and the dependent resource JAR files are packaged as web applications in an EAR file. This EAR file is deployed to an Application server. The client machine with JWS installed uses a web browser to connect to the URL of the client application JNLP descriptor file to download and run the client application.

Using Java Web Start from Java SE Runtime Environment 6.0 or later is highly recommended. All the platforms supported by the application client for WebSphere Application Server are supported.

You can use the following:

- Java Web Start on the Java Standard Edition Developer Kits that IBM provides, packaged in Application Client for WebSphere Application Server
- Java Web Start on Java SE 6 Development Kit or Java SE Runtime Environment 6.0, which you can download from the Oracle website for Windows, Linux and Solaris operating systems
- Java Web Start on HP-UX JDK or JRE for Java Platform, Standard Edition, Version 6, which you can download from the HP website

buildClientLibJars tool:

For a Java Platform, Enterprise Edition (Java EE) application client application and or Thin application client application to be launched using Java Web Start (JWS), the properties files bundled in Application Client for WebSphere Application Server must be installed in the Java Web Start. Use this tool to create those property JAR files. The Java Web Start client is used with platforms that support a web browser.

The buildClientLibJars tool copies the JAR files from the Application Client for WebSphere Application Server installation and creates a `properties.jar` file, which contains the properties files from the Application Clients installation properties directory to a specified location. When this property is created, the tool uses the value of `keystore`, `storepass`, `alias` and `storetype` to sign all of the JAR files in the specified location.

```
Windows usage: buildClientLibJars.bat  [-help] [-verbose] destdir keystore storepass alias storetype

Unix usage:    buildClientLibJars.sh  [-help] [-verbose] destdir keystore storepass alias storetype
```

where:

- `-help` will display the message
- `-verbose` will turn on verbose message
- `destdir` will output the destination directory name
- `keystore` is the key store file
- `storepass` is the key store password
- `alias` key is the alias name
- `storetype` is the key store type

Client application Java Network Launcher Protocol deployment descriptor file

The deployment descriptor file is the main Java Network Launcher Protocol (JNLP) descriptor file for the client application.

Location

The client application has an Application Clients runtime dependency that provides the following:

- Java SE Runtime Environment from IBM
- Application Clients run-time properties
- SSL KeyStore and TrustStore file
- Application Clients run-time library JAR files (optional for Thin Application client applications)

If the Application Clients run-time dependency is not met, it is downloaded and installed in Java Web Start (JWS), as described by the Application Clients run-time installer JNLP descriptor file. For example:

```
<j2se version="1.6" href="http://your_server.com/jws/wasappclient/download.jnlp"/>
```

Usage notes

The client application must also include the `WebSphereClientLauncher.jar` file, which contains the launcher class, com.ibm.websphere.client.launcher.ClientLauncher, that completes one of the following actions:

- If it is a Java Platform, Enterprise Edition (Java EE) Application client application (that is the resources for the application contain an EAR file with a client application), the EAR file must be specified as a JAR resource so that it can be downloaded to JWS and specified in the system property, `com.ibm.websphere.client.launcher.ear`. See "JNLP descriptor file for a Java EE Application client application" for an example.
- If it is a Thin Application client application, the Thin Application client application JAR file must be specified as a JAR resource so that it can be downloaded to JWS and the name of the class containing main method entry point is specified in the system property, `com.ibm.websphere.launcher.main`. See "JNLP descriptor file for a Thin Application client application" on page 2026 for an example.

The JNLP specification requires all the resource (JAR or EAR) files used in a JNLP file to be signed.

You can specify the –CC arguments defined in the launchClient tool for a J2EE Application client application in application arguments section of the JNLP descriptor files. However, only –CCD is supported for a Thin Application client application to define system properties and the JNLP <property> tag can also be used to define system properties. See the following example for details:

```
<property name="java.naming.provider.url" value="corbaloc:iiop:myserver.com:9089"/>
```

For a J2EE Application client application, specify the following application arguments as defined in the JNLP.

1. Specify your target server provider URL, as shown in the following example:

   ```
   <argument> >-CCDjava.naming.provider.url =corbaloc:iiop:myserver.mydomain.com:9080 </argument>
   ```

2. Specify the SSL Key File and SSL Trust File location. These files are expected to be available in the client machine. To use the ones in the Application Clients run-time dependency installed in JWS cache, specify these application arguments:

   ```
   <argument> -CCDcom.ibm.ssl.keyStore=${WAS_ROOT}/etc/key.p12 </argument>
   <argument> -CCDcom.ibm.ssl.trustStore=${WAS_ROOT}/etc/trust.p12 </argument>
   ```

3. Specify the initial naming context factor, as shown in the following example:

   ```
   <argument>-CCDjava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory </argument>
   ```

   For a Thin Application client application, you also need to specify the actual location of the `sas.client.props` and `ssl.client.props` files located in the Application Clients runtime dependency that is installed in the JWS cache.

   ```
   <argument>-CCDcom.ibm.CORBA.ConfigURL=file:${WAS_ROOT}/properties/sas.client.props </argument>
   <argument>-CCDcom.ibm.SSL.ConfigURL=file:${WAS_ROOT}/properties/ssl.client.props </argument>
   ```

   If any of the default settings in the `sas.client.props` and `sas.client.props` file need modifying, use the –CCD to change the settings through the system properties, as shown in the following example:

   ```
   <argument>-CCDjavacom.ibm.CORBA.securityEnabled=false </argument>
   ```

   Important: The ${WAS_ROOT} token used in the JNLP file is replaced by the launcher class, com.ibm.websphere.client.launcher.ClientLauncher, to the actual location of the Application Clients run-time dependency installation in the JWS cache. If you are using JSP to dynamically create this JNLP description file, you must escape this token because it has a different meaning in JSP 2.0. See the following example for details:

   ```
   <argument>-CCDcom.ibm.ssl.keyStore=\${WAS_ROOT}/etc/key.p12 </argument>
   <argument>-CCDcom.ibm.ssl.trustStore=\${WAS_ROOT}/etc/trust.p12 </argument>
   ```

JNLP descriptor file for a Java EE Application client application:

The deployment descriptor file is the main Java Network Launcher Protocol (JNLP) descriptor file for the client application.

Here is an example of the client application JNLP descriptor file for a Java EE Application client application:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This sample program is provided AS IS and may be used, executed, copied and modified
without royalty payment by customer (a) for its own instruction and study, (b) in order
to develop applications designed to run with an IBM WebSphere product, either for customer's
own internal use or for redistribution by customer, as part of such an application, in
customer's own products.

Licensed Materials - Property of IBM

5724-I63, 5724-H88, 5724-H89, 5655-N02, 5724-J08

Copyright IBM Corp. 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with
IBM Corp.
-->

<jnlp spec="1.0+" codebase="http://your_server:port_number/jws/wasappclient/apps/">
 <information>
  <title>Java EE Client Example</title>
  <vendor>IBM</vendor>
  <homepage href="null"/>
  <description>Java WebStart example: Launching Java EE Application Client</description>
  <description kind="short">Java EE Applicaiton Client</description>
  <description kind="tooltip">Java EE Application Client</description>
 </information>

 <security>
  <all-permissions/>
 </security>

 <resources>
  <j2se href="http://your_server:port_number/jws/wasappclient/JREDownload.xjnlp" version="1.6"/>
  <jar href="../lib/WebSphereClientLauncher.jar" download="eager" main="false"/>
  <jar href="../lib/properties.jar" download="eager" main="false"/>
  <jar href="SwingCalculator.ear" download="eager" main="false"/>

  <property name="com.ibm.websphere.client.launcher.ear" value="SwingCalculator.ear"/>
 </resources>

 <application-desc main-class="com.ibm.websphere.client.launcher.ClientLauncher">
  <argument>-CCproviderURL=corbaloc:iiop:tiu03.torolab.ibm.com:2809</argument>
 </application-desc>

</jnlp>
```

JNLP descriptor file for a Thin Application client application:

The deployment descriptor file is the main Java Network Launcher Protocol (JNLP) descriptor file for the client application. If it is a Thin Application client application, then the launcher class uses the current JVM from the Application Clients run-time dependency and invokes the Thin Application client application main method.

Here is an example of the JNLP descriptor file for a Thin Application client application.

```
 This sample program is provided AS IS and may be used, executed, copied and modified
without royalty payment by customer (a) for its own instruction and study, (b) in order
to develop applications designed to run with an IBM WebSphere product, either for customer's
own internal use or for redistribution by customer, as part of such an application, in
customer's own products.

Licensed Materials - Property of IBM
```

```
5724-I63, 5724-H88, 5724-H89, 5655-N02, 5724-J08

Copyright IBM Corp. 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with
IBM Corp.

Licensed Materials - Property of IBM

5724-I63, 5724-H88, 5724-H89, 5655-N02, 5724-J08

Copyright IBM Corp. 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with
IBM Corp.
-->

<!-
=====================================================================
->
<!-- TODO: change "codebase" to the actual URL location of the jnlp file ->
=====================================================================
->
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+"
codebase="http://your_server:port_number/jws/wasappclient/apps">
  <information>
  <title>Thin Base Calculator Client Samples</title>
  <vendor>IBM</vendor>
  <description>Thin Base Calculator Client Samples</description>
  <offline-allowed/>
 </information>

 <security>
  <all-permissions/>
 </security>
  <resources>
   <j2se version="1.6" href="http://your_server:port_number/jws/wasappclient/JREDownload.xjnlp"/>

   <jar href="/jws/wasappclient/lib/WebSphereClientLauncher.jar" main="true"/>
   <jar href="BasicCalculatorClientCommon.jar"/>
   <jar href="BasicCalculatorEJB.jar"/>
   <jar href="BasicCalculatorThinClient.jar"/>

   <property name="com.ibm.websphere.client.launcher.main"
      value="com.ibm.websphere.samples.technologysamples.basiccalcthinclient.BasicCalculatorClientThinMain"/>
   <property name="java.naming.factory.initial"
              value="com.ibm.websphere.naming.WsnInitialContextFactory" />
   <property name="java.naming.provider.url"
               value="corbaloc:iiop:tiu03:2809"/>

   </resources>

   <add</argument>
    <argument>1</argument>
    <argument>2</argument>
   </application-desc>
</jnlp>
```

ClientLauncher class:

The class, com.ibm.websphere.client.installer.ClientLauncher, contains a main() method that is called by Java Web Start (JWS) to launch the client application. The Java Web Start client is used with platforms that support a web browser.

This client is packaged in the `WebSphereClientLauncher.jar` file that is located in the Application Client for WebSphere Application Server installation under the <*app_client_root*>/ lib/webstart directory.

The launcher class requires that the following properties are defined. These properties are not defined in a separate properties file. Instead, the properties are defined as part of the Java Network Launching Protocol (JNLP) files.

com.ibm.websphere.client.launcher.main
If the client application is a Thin Application client, then this property should be specified. It specifies the class where the main entry point of the client application resides.

com.ibm.websphere.client.launcher.ear
If the client application is a Java Platform, Enterprise Edition (Java EE) Application client, then this property should be specified. It specifies the name of the EAR file to be executed. This property takes precedence over com.ibm.websphere.client.launcher.main. However, only one of the two properties should be specified.

Application client launcher for Java Web Start:

The application client launcher for Java Web Start is a Java class, com.ibm.websphere.client.installer.ClientLauncher, which has a main() method that Java Web Start calls to start the application client container and to invoke the application client's main() method. It provides similar functions as the **`lauchClient`** command line tool to start application clients from the command line.

The com.ibm.websphere.client.launcher.ClientLauncher class is packaged in the WebSphereClientLauncher.jar file under the <app_client_root>/lib/webstart directory.

The launcher tool requires that the following properties are defined.

com.ibm.websphere.client.launcher.main
If the client that is to be run is a thin client, then this property should be specified. It specifies the class where the main entry point of the application resides. It is the main class name for a Thin application client. If it is set, the launcher will not start the client container, it will rather invoke the main method for the application directly. However, if `com.ibm.websphere.client.launcher.ear` is also set, it will be ignored.

com.ibm.websphere.client.launcher.ear
If the client that is to run is the Java Platform, Enterprise Edition (Java EE) client, then this property should be specified. It specifies the name of the ear file to be executed. This property takes precedence over `com.ibm.websphere.client.launcher.main` although only one of the two properties should be specified.

These properties are not defined in a separate properties file. Instead, they are defined as part of the Java Network Launching Protocol files.

When `com.ibm.websphere.client.launcher.ear` is set, the application client launcher for JWS supports almost all of the –CC arguments as the **`lauchClient`** command line tool supports. However, if only `com.ibm.websphere.client.launcher.main` is set, the launcher will only support the –CCD argument. The following table shows the comparison of the supported –CC arguments for the **`launchClient`** command line tool and the application client launcher for JWS:

Table 270. Comparison of the supported –CC arguments for the **`launchClient`** *command line tool and the application client launcher for JWS. Comparison of the supported –CC arguments*

-CC argument	launchClient	Application client launcher for JWS
-CCverbose	Yes	Yes
-CCjar	Yes	Yes
-CCclasspath	Yes	N/A

Table 270. Comparison of the supported –CC arguments for the **launchClient** *command line tool and the application client launcher for JWS (continued). Comparison of the supported –CC arguments*

-CC argument	launchClient	Application client launcher for JWS
-CCadminConnectorHost	Yes	Yes
-CCadminConnectorPort	Yes	Yes
-CCadminConnectorType	Yes	Yes
-CCadminConnectorUser	Yes	Yes
-CCaltDD	Yes	Yes
-CCBootstrapHost	Yes	Yes
-CCBootstrapPort	Yes	Yes
-CCproviderURL	Yes	Yes
-CCinitonly	Yes	N/A
-CCtrace	Yes	Yes
-CCtracefile	Yes	Yes
-CCsecurityManager	Yes	N/A
-CCsecurityMgrClass	Yes	N/A
-CCsecurityMgrPolicy	Yes	N/A
-CCD	Yes	Yes
-CCexitVM	Yes	Yes
-CCdumpJavaNameSpace	Yes	Yes
-CCsoapConnectorPort	Yes	Yes
-CCtraceMode	Yes	Yes
-CCclassLoaderMode	Yes	Yes

Macro expansion is supported for the –CCD argument by the application client launcher for JWS. The launcher will automatically substitute certain macro keys (enclosed with ${...}) with the calculated value at runtime. For example, if a macro key is used in the –CCD argument in the application client JNLP manifest file,

```
<argument>-CCDcom.ibm.ssl.keyStore= ${WAS_ROOT}/etc/key.p12</argument>
```

it will be expanded to the JWS cache installation root location and the argument will become:

```
-CCDcom.ibm.ssl.keyStore=/home/tiu/.java/deployment/cache/javaws/ext/E1134532441112/etc/key12.p12
```

The following table shows the three macro keys that are currently supported and will be substituted by the launcher:

Table 271. Currently supported macro keys. Supported macro keys

Macro key	Value
${WAS_ROOT}	Installation root location within the JWS cache that is used by the application client container and runtime installer for JWS.
${JAVA_HOME}	Location of Java home. The return value of System.getProperty(“java.home”).
${USER_HOME}	Location of user home. The return value of System.getProperty(“user.home”).

Preparing the application client run time dependency component for Java Web Start

To launch a Java Platform, Enterprise Edition (Java EE) application client application, a Thin application client application, or both using Java Web Start (JWS), a Java Runtime Environment implementation Java archive (JAR) that IBM provides, the library JAR files and properties files bundled in Application Client for WebSphere Application Server must be installed in the JWS. Learn the steps to build the application client run time dependency component from an application client installation. It is packaged as a web application archive (WAR) file that can be installed in an application Server.

Before you begin

Install the Application Client for WebSphere Application Server for the operating system to which the client application deploys. If there is a requirement to deploy the client application to multiple operating systems, the application client run time dependency component must be built separately for each operating system that client application supports.

Procedure

1. Install the Application Client for WebSphere Application Server for the client application supported operating systems.
2. Change the directory to the installation bin directory.
3. Run the “buildClientRuntime tool” on page 2032 to generate the application client run time JAR file, which contains the Java Standard Edition Runtime Environment, the run time library JAR files, properties files, and the SSL KeyStore and TrustStore files from the application client installation.
4. Run the buildClientLibJars tools to package up the properties files in the properties directory of the application client installation into a properties.jar file in the specified location. The buildClientLibJars tools will also copy the `WebSphereClientLauncher.jar` file and `WebSphereClientRuntimeInstaller.jar` file from the application client installation to the specified location. All jar files in the specified location will be signed by the provided certificate.

   For example, if you are using Version 7.0 and using the test certificate that is included in the application client installation:

   ```
   buildClientLibJars C:\Temp\webstart ..\etc\DummyClientKeyFilejar WebAS "websphere dummy client" JKS
   ```
5. Create a JavaServer Pages (JSP) file or use a servlet to generate the application client run time installer Java Network Launching Protocol (JNLP) descriptor to respond to Java Web Start request. See the Java Web Start deployment sample in the application client installation.
6. Package the two signed JAR files, `WASClient7.0_windows.jar` and `WebSphereClientRuntimeInstaller.jar`, and the JSP file or servlet for generating the Application Client run time installer JNLP descriptor into a web application archive (WAR) file. This WAR file is packaged into an EAR file that can be deployed to an application server. See the Java Web Start deployment sample in the application client installation.

Results

Your web application is ready to serve the application client run time and the JRE environment.

Example

```
<!-- This sample program applies to WebSphere Application Server, Version 6.1.
It is provided AS IS and may be used, executed, copied and modified
without royalty payment by customer (a) for its own instruction and study, (b) in order
to develop applications designed to run with an IBM WebSphere product, either for customer's
own internal use or for redistribution by customer, as part of such an application, in
customer's own products.

Product 5630-A36,  (C) COPYRIGHT International Business Machines Corp., 2005
All Rights Reserved * Licensed Materials - Property of IBM
-->
```

```
<%-- // to set the Last_Modified header so that the JNLP client will know whether to download
     // the JNLP file again and update the cached copy.
     String jspPath = application.getRealPath(request.getServletPath());
     java.io.File jspFile = new java.io.File(jspPath);
     long lastModified = jspFile.lastModified();
%><%
      // locally declared variables
      String url=request.getRequestURL().toString();
      String jnlpCodeBase=url.substring(0,url.lastIndexOf('/'));
      String jnlpRefURL=url.substring(url.lastIndexOf('/')+1,url.length());

      // Need to set a JNLP mime type - if WebStart is installed on the client,
      // this header will induce the browser to drive the WebStart Client
      response.setContentType("application/x-java-jnlp-file");                              1
      response.setHeader("Cache-Control", null);
      response.setHeader("Set-Cookie", null);
      response.setHeader("Vary", null);
      response.setDateHeader("Last-Modified", lastModified);

      // An installer must reply with the version number for a given install
      if (response.containsHeader("x-java-jnlp-version-id"))
        response.setHeader("x-java-jnlp-version-id", "WASClient6.1.0");               2
      else
        response.addHeader("x-java-jnlp-version-id", "WASClient6.1.0");
%>

<?xml version="1.0" encoding="utf-8"?>

<!-- ========================================================== -->
<!-- TODO: change "codebase" to the actual url location of this jsp -->
<!-- ========================================================== -->

<jnlp spec="1.0+"
codebase="http://YOUR_APP_SERVER:PORTNUMBER/WEBAPP_CONTEXT_ROOT/Runtime/WebSphereJre">

<information>
  <title>Application Client Java Runtime Environment</title>
  <vendor>IBM</vendor>
  <icon href="icon.gif"/>
  <description>Application Client Java Runtime Environment</description>
  <description kind="short">Application Client JRE</description>
  <description kind="tooltip">Application Client JRE</description>
  <offline-allowed/>
</information>

<security>
 <all-permissions/>
</security>

 <resources>
  <j2se version="1.4+"/><%-- The installer can use any 1.4 JRE --%> 3
   <jar href="WebSphereClientRuntimeInstaller.jar" main="true"/> 4

  <!-- JRE version registration with Web Start -->
  <property name="com.ibm.websphere.client.jre.version" value="WASClient6.1.0"/> 5
 </resources>

 <resources os="Windows"> 6
<!-- ========================================================== -->
<!-- TODO: the property value for unix platform is "java/jre/bin/javaw" -->
<!--   and the "os" value match to your target client machine platform             -->
<!-- ========================================================== -->
```

```
   <jar href="WASClient6.1.0_Windows.jar"/> 7

<!-- ============================================================ -->
<!-- TODO:  property value for unix platform is "java/jre/bin/javaw" -->
<!-- ============================================================ -->
<!-- relative path of the jre executable —->

   <property name="com.ibm.websphere.client.jre.launch.java"
 value="java\jre\bin\javaw.exe"/> 8

 </resources>
 <installer-desc main-class="com.ibm.websphere.client.installer.ClientRuntimeInstaller"/>
</jnlp>
```

1. Specifies that the file is a JNLP mime type so that the browser can process the JNLP file.
2. Specifies the exact version of this Application Client run time dependency component in the response by setting the HTTP header field: `x-java-jnlp-version-id`.
3. Specifies the required JRE version to run the installer program.
4. Specifies the installer `WebSphereClientRuntimeInstaller.jar` file, which contains the ClientRuntimeInstaller class.
5. Specifies a system property that defines the version of Application Client run time dependency component. This version is registered to the JNLP client.
6. Specifies resources for a particular platform. Each supported client application platform needs its own separate JAR file.
7. Specifies the Application Client run time dependency component JAR file.
8. Specifies the program to call that starts a JVM for the client application.

buildClientRuntime tool:

For a Java Platform, Enterprise Edition (Java EE) application client application and or Thin application client application to be launched using Java Web Start (JWS), the library JAR files bundled in Application Client for WebSphere Application Server must be installed in the Java Web Start. Use this tool to build those JAR files. The Java Web Start client is used with platforms that support a web browser.

The buildClientRuntime tool builds the required components from the WebSphere Application Server clients installation into the JAR file specified on the command. This JAR file contains:

- License files
- Java SE Runtime Environment 6 (JRE 6) that IBM provides
- Application Clients runtime properties and configuration
- SSL KeyStore and TrustStore files
- Runtime library JAR files

In the case of building an Application Clients runtime JAR file only for serving Thin Application client applications and not for Java EE Application client applications, the runtime library JAR files and the Application Clients runtime properties files are not included, except the configuration files, `sas.client.props`, `ssl.client.props` and `soap.client.props`, located in the `WAS_ROOT/properties` directory. The Java Web Start client is used with platforms that support a web browser.

The command-line invocation syntax for the buildClientRuntime tool is shown in the following example:

```
Windows Usage:    buildClientRuntime.bat  [-help] [-verbose] outfile keystore storepass alias storetype

Unix Usage:         buildClientRuntime.sh    [-help] [-verbose] outfile keystore storepass alias storetype
```

where:

- -help will display the message
- -verbose will turn on verbose message
- outfile is the output file name
- keystore is the key store file
- storepass is the key store password
- alias is the key alias name
- storetype is the key store type

ClientRuntimeInstaller class:

This section provides information on the ClientRuntimeInstaller class.

This class, com.ibm.websphere.client.installer.ClientRuntimeInstaller, contains a main() method that Java Web Start (JWS) calls to install the Application Client for WebSphere Application Server run-time dependency component in JWS cache. It is packaged in `WebSphereClientRuntimeInstaller.jar` file located in the Application Client for WebSphere Application Server installation in the *<app_server_root>*/`JWS` directory.

Specify the `WebSphereClientRuntimeInstaller.jar` file and the Application Client run-time dependency component JAR file as JAR resources in the Application Client run-time installer Java Network Launcher Protocol (JNLP) descriptor file. See the following example for details:

```
<jar href="Launcher/WebSphereClientRuntimeInstall.jar" main="true"/>
<jar href="Launcher/WASClient6.1_windows.jarRuntimeInstall.jar" main="true"/>
```

The ClientRuntimeInstaller class main method requires the following properties to be set in the JNLP file:

com.ibm.websphere.client.jre.version
Specifies a Java Runtime Environment (JRE) version name that is to be used when referring to the Application Client run-time dependency component.

com.ibm.websphere.client.jre.launch.java
Specifies the relative location of the javaw.exe program in the Application Client run-time dependency component JAR file.

The previously mentioned properties, JRE version name and the location of the javaw.exe program are registered to the Java Web Start Application Manager, as shown in the following example:

```
<property name="com.ibm.websphere.client.jre.version" value="WASclient6.1"/>
<property name="com.ibm.websphere.client.jre.launch.java" value="java\jre\bin\javaw.exe"/>
```

Using the Java Web Start sample

The EAR file, `WasAppClientRuntime.ear`, is provided in the *app_client_root*/`samples/bin/WasAppClientRuntime` directory of the Client Application for WebSphere Application Server installation. This EAR file provides a sample Application Clients run-time installer JNLP descriptor file and a sample Application Clients run-time library component JNLP descriptor file. Follow the steps in this task to build the Application Clients run-time dependency component and the Application Clients run-time library component. Add these components to the `WebSphereClientRuntime.ear` file, and then install the EAR file in an Application Server to be used by the client application.

About this task

There is a new Java Web Start sample available in the client sample gallery for WebSphere Application Server V7.0. Refer to the client sample gallery in the Application Client for WebSphere Application Server product. The name of the new sample is “Java Web Start Deployment Sample”.

Installing Java Web Start

Learn about the steps that are necessary to install Java Web Start (JWS).. Java Web Start technology is provided by the Java SE runtime environment to deploy Java EE application clients (including Thin application clients) on the remote client machine with a single click from a web browser on the client machine.

Before you begin

Before you begin this task, see the “Preparing the application client run time dependency component for Java Web Start” on page 2030 topic to understand Java Web Start (JWS) technology and components.

Note: The Sun Java Web Start, which is available from Sun Microsystems, is not compatible with the IBM Runtime Environment, Java 2 Technology Edition, which is provided by WebSphere Application Server and the IBM Application Client. The IBM Runtime Environment contains some additional functionality that is not supported in the Sun Java Web Start. Also, the IBM Runtime Environment uses a different packaging structure than the Sun Java Web Start. Use the IBM Runtime Environment.

About this task

Complete the following steps to install JWS:

Procedure

1. Install IBM Application Client for WebSphere Application Server.
2. Change your directory to the `javaws` path.
   - *client_install_root*`\java\jre\lib\javaws`
3. Run the update settings script from the path mentioned in the previous step.
   - Run the `updateSetting.sh` script
4. Change your path to the JWS installed path. For example, enter:
   - *client_install_root*`\java\jre\javaws`
5. Run `javaws` from the path mentioned in the previous step.
   - Run `./javaws` command.

Using a static JNLP file with Java Web Start for Application clients

Do not use JSP to dynamically generate a JNLP file, otherwise the JNLP jsp page cannot be opened in some IE browsers.

About this task

To use a static JNLP file, you will need to add the following mime type mapping in the web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
       <display-name>
       WAS Client runtime for Java Web Start</display-name>
       <welcome-file-list>
              <welcome-file>index.html</welcome-file>
              <welcome-file>index.htm</welcome-file>
              <welcome-file>index.jsp</welcome-file>
              <welcome-file>default.html</welcome-file>
              <welcome-file>default.htm</welcome-file>
              <welcome-file>default.jsp</welcome-file>
       </welcome-file-list>
       <mime-mapping>
```

```
        <extension>jnlp</extension>
        <mime-type>application/x-java-jnlp-file</mime-type>
        </mime-mapping>
</web-app>
```

Running the IBM Thin Client for Enterprise JavaBeans (EJB)

An EJB Client is a Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) Java Platform, Standard Edition (Java SE) application that accesses remote Enterprise Java Beans from a server through Java Naming and Directory Interface (JNDI) look up. IBM Thin Client for EJB offers a smaller footprint and is easy to deploy to a Java SE environment and an Eclipse Rich Client Platform (RCP) environment. You can bundle the IBM Thin Client for EJB library using the WebSphere Application Server installation or the Application Client for WebSphere Application Server installation with your application. The IBM Thin Client for EJB also extends the choice of Java SE runtime. It can be run in the Java Runtime Environment (JRE) that is packaged with the WebSphere Application Server product, the JRE that is downloaded from the Oracle website, or the JRE that is downloaded from the HP website.

Before you begin

The IBM Thin Client for EJB can access version 2.x and version 3.x EJB on the WebSphere Application Server using the JNDI lookup, but it cannot access version 3.x EJB through resource injection. Resource injection is supported if the client application is a Java Platform, Enterprise Edition (Java EE) Application Client running within the Java Platform, Enterprise Edition (Java EE) Application Client Container.

Before you set up an EJB Thin Client environment, obtain the Java archive (JAR) file for the EJB Thin Client for WebSphere Application Server. To obtain the EJB Thin Client for WebSphere Application Server, install WebSphere Application Server or Application Client. The EJB Thin Client for WebSphere Application Server file, `com.ibm.ws.ejb.thinclient.zos_8.5.0.jar`, is located in the *app_server_root*`\runtimes` directory. `Use the com.ibm.ws.ejb.thinclient.zos_8.5.0.jar` for any client that is running on z/OS. Determining the client to use depends on the client platform, and not the platform of the server you are connecting to.

Copy the Java archive (JAR) file for the IBM Thin Client for EJB with WebSphere Application Server product, `com.ibm.ws.ejb.thinclient.zos_8.5.0.jar` and the `endorsed_apis_8.5.0.jar` files, to other machines to create a lightweight client environment that enables communications with the products. Copies of the IBM Thin Client for EJB are subject to the same terms and conditions of the license agreement for the WebSphere product where you obtained the Thin Client for EJB. Refer to the license agreements for correct usage and other limitations.

Copy the *app_server_root*`\runtimes\endorsed\endorsed_apis_8.5.0.jar` file into the default directory, `JAVA_JRE\lib\endorsed`. Alternatively, you can use the java.endorsed.dirs property to specify a directory of your choice. If you choose to use an alternative directory, it is a best practice to only include the endorsed_apis JAR file.

Important: The Pluggable Application Client is deprecated. It is replaced by the IBM Thin Client for EJB.

Attention: When running the IBM Thin Client for EJB, and the `-Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager` command line option is used, a ClassDefNotFoundError error is thrown. The use of WsLogManager is not supported in the IBM Thin Client for EJB, but you can use another Java logging manager.

About this task

Run the IBM Thin Client for EJB, by completing the following steps.

Procedure

1. Invoke the client application. Run the following Java command:

   Add the following system properties to the Java command if you want authentication and SSL enabled:

   ```
   export LIBPATH=<app_server_root>/lib:$LIBPATH
   <java_install_root>/bin/java
   -classpath com.ibm.ws.ejb.thinclient.zos_8.5.0.jar:<list_of_your_application_jars_and_classes>
   -Djava.naming.provider.url=iiop://<your_application_server_machine_name>
   -Dcom.ibm.SSL.ConfigURL=file:///home/user1/ssl.client.props
   -Dcom.ibm.CORBA.ConfigURL=file:///home/user1/sas.client.props
   <fully_qualified_class_name_to_run>
   ```

2. Provide IIOP authentication configuration and Client SSL Configuration. Add the following system properties to the Java command:

   ```
   -Dcom.ibm.SSL.ConfigURL=file:///home/user1/ssl.client.props
   -Dcom.ibm.CORBA.ConfigURL=file:///home/user1/sas.client.props
   ```

   You can obtain the `ssl.client.props` file and `sas.client.props` file from the WebSphere Application Server installation and modify the file to suit your environment. You must, at a minimum, update the location of the key files in the `ssl.client.props` file to the match location of your target environment. For example,

   ```
   -Dcom.ibm.ssl.keyStore=/home/user1/etc/key.p12
   -Dcom.ibm.ssl.trustStore=/home/user1/etc/trust.p12
   ```

3. Run your client application:

   - Enter the following command if you have copied the endorsed_apis_8.5.0.jar file into the `JAVA_JRE\lib\endorsed` default directory; for example:

     ```
     %JAVA_HOME%\bin\java -Dcom.ibm.SSL.ConfigURL=file:\\\home\sample\ssl.client.props <your_client_application>
     ```

   - Enter the following command if you have copied the endorsed_apis_8.5.0.jar file into a directory other than the default `JAVA_JRE\lib\endorsed` directory; for example:

     ```
     %JAVA_HOME%\bin\java
     -Djava.endorsed.dirs=<directory_that_includes_endorsed_apis_8.5.0.jar>
     -Dcom.ibm.SSL.ConfigURL=file:\\\home\sample\ssl.client.props <your_client_application>
     ```

What to do next

Enable trace for the IBM Thin Client for EJB by adding the following to the Java command.

```
-Dcom.ibm.ejs.ras.lite.traceSpecification=*=all
```

Running Java thin client applications

You can run Java thin client applications on machines installed with either a WebSphere Application Client installation or a WebSphere Application Server installation.

About this task

Important: Java thin clients are not packaged with JDBC provider classes. For example, the WebSphere Application Server Version 7.0 Java thin client is not packaged with Apache Derby 10.2 classes. Likewise, the version 6.1 Java thin client is not packaged with Cloudscape Version 5.1, Cloudscape Version 10.0, or Cloudscape version 10.1 classes. Therefore, to utilize the JDBC provider classes (such as Apache Derby, Oracle, DB2, Informix, or Sybase) on a Java thin client, you must:

1. Add the classes to your Java thin client application environment.
2. Make the classes visible to the Java thin client application. To do this, add the path to the classes in the client classpath within the script that launched the client program.

Otherwise, any attempt to load a database class (such as through the JNDI lookup of a datasource) results in a ClassNotFoundException.

The Java invocation to run a Java thin client application varies between a client and a server. If your Java thin client application needs to run on both a client installation and a server installation, follow the steps in the Running a Java thin client application on a server machine topic.

Procedure

- “Running a Java thin client application on a client machine” on page 2038
- “Running a Java thin client application on a server machine” on page 2039

Example

Your Java thin application client no longer needs additional code to set security providers if you have enabled security for your WebSphere Application Server instance. This code found in IBM i Java thin or pluggable application clients should be removed to prevent migration and compatibility problems. The java.security file from your WebSphere instance in the properties directory is now used to configure the security providers.

- Running the thin or pluggable application client with security enabled

Running the thin or pluggable application client with security enabled. The following code examples illustrates how security providers were set programmatically in the main() method and occurred prior to any code that accessed enterprise beans:

```
import java.security.*;
  ...
  if (System.getProperty("os.name").equals("OS/400")) {

    // Set the default provider list first.
    Provider jceProv = null;
    Provider jsseProv = null;
    Provider sunProv = null;

    // Allow for when the Provider is not needed, when
    // it is not in the client application's classpath.
    try {
      jceProv = new com.ibm.crypto.provider.IBMJCE();
    }
    catch (Exception ex) {
 ex.printStackTrace();
    throw new Exception("Unable to acquire provider.");
 }

    try {
      jsseProv = new com.ibm.jsse.JSSEProvider();
    }
    catch (Exception ex) {
 ex.printStackTrace();
    throw new Exception("Unable to acquire provider.");
 }

    try {
      sunProv = new sun.security.provider.Sun();
    }
    catch (Exception ex) {
 ex.printStackTrace();
    throw new Exception("Unable to acquire provider.");
 }

    // Enable providers early and ahead of other providers
    // for consistent performance and function.
    if ( (null != sunProv) && (1 != Security.insertProviderAt(sunProv, 1)) ) {
      Security.removeProvider(sunProv.getName());
      Security.insertProviderAt(sunProv, 1);
    }
    if ( (null != jceProv) && (2 != Security.insertProviderAt(jceProv, 2)) ) {
```

```
      Security.removeProvider(jceProv.getName());
      Security.insertProviderAt(jceProv, 2);
    }
    if ( (null != jsseProv) && (3 != Security.insertProviderAt(jsseProv, 3)) ) {
      Security.removeProvider(jsseProv.getName());
      Security.insertProviderAt(jsseProv, 3);
    }

    // Adjust default ordering based on admin/startstd properties file.
    // Maximum allowed in property file is 20.
    String provName;
    Class  provClass;
    Object provObj =  null;

    for (int i = 0; i < 21; i++) {
      provName = System.getProperty("os400.security.provider."+ i);

      if (null != provName) {

        try {
          provClass = Class.forName(provName);
          provObj = provClass.newInstance();
        }
        catch (Exception ex) {
          // provider not found
          continue;
        }

        if (i != Security.insertProviderAt((Provider) provObj, i)) {

          // index 0 adds to end of existing list
          if (i != 0) {
            Security.removeProvider(((Provider) provObj).getName());
            Security.insertProviderAt((Provider) provObj, i);
          }
        }
      } // end if (null != provName)
    } // end for (int i = 0; i < 21; i++)
  } // end if ("os.name").equals("OS/400")
```

For examples of Java thin client applications, refer to the Samples section of the information center.

Running a Java thin client application on a client machine

To run a Java thin client application on a machine with Application Client for WebSphere Application Server installed, use the setup Client command then start the application.

Before you begin

Before performing this task, you must install the Java thin application client from the Application Client for WebSphere Application Server installation.

Procedure

1. Set up the client application environment. Run the setupClient command.

   app_client_root/AppClient/bin/setupClient.sh

2. Run a Java command to invoke your client application.

   ```
   $JAVA_HOME/bin/java $WAS_LOGGING
   -classpath "$WAS_CLASSPATH: <list_of_your_application_jars_and_classes>"
   -Djava.ext.dirs=$JAVA_JRE/lib/ext:$WAS_EXT_DIRS:$WAS_HOME/plugins:$WAS_HOME/lib/WMQ/java/lib"
   -Djava.naming.provider.url=iiop://<your_application_server_machine_name>
   -Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
   $SERVER_ROOT $CLIENTSAS $CLIENTSSL <fully_qualified_class_name_to_run>
   ```

Running a Java thin client application on a server machine

To run a Java thin client application on a machine with WebSphere Application Server installed, use the setupClient command then start the application.

Before you begin

You must install WebSphere Application Server before performing this task.

Procedure

1. Set up the Thin application client environment.

   Use the setupCmdLine shell.

   profile_root/bin/setupCmdLine.sh

2. Run the application client.

   Perform one of the following methods:

   - Run a Java command to call your main class directly.

     ```
     "$JAVA_HOME/bin/java" $WAS_LOGGING
     -Djava.security.auth.login.config="$USER_INSTALL_ROOT/properties/wsjaas_client.conf"
     -Djava.ext.dirs="$JAVA_HOME/jre/lib/ext:$WAS_EXT_DIRS:$WAS_HOME/plugins: $WAS_HOME/lib/WMQ/java/lib"
     -Djava.naming.provider.url=<an_IIOP_URL_or_a_corbaloc_URL_to_your
     application_server_machine_name>
     -Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
     -Dserver.root="$WAS_HOME" $USER_INSTALL_PROP "$CLIENTSAS" "$CLIENTSSL"
     -classpath "$WAS_CLASSPATH:<list_of_your_application_jars_and_classes>"
     <fully_qualified_class_name_to_run> <your_application_parameters>
     ```

     For more information on IIOP and corbaloc URLs, see Developing applications that use JNDI.

   - Enter a command to use the WebSphere Application Server launcher.

     ```
     "$JAVA_HOME/bin/java" $WAS_LOGGING
     -Djava.security.auth.login.config="$USER_INSTALL_ROOT/properties/wsjaas_client.conf"
      "-Dws.ext.dirs=<list_of_your_application_jars_and_classes>
     $WAS_EXT_DIRS:$WAS_USER_DIRS"
     -Djava.naming.provider.url=<an_IIOP_URL_or_a_corbaloc_URL_to_your
     application_server_machine_name>
     -Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
     "-Dserver.root=$WAS_HOME"
     "$CLIENTSAS" "$CLIENTSSL" $USER_INSTALL_PROP -classpath "$WAS_CLASSPATH"
     com.ibm.ws.bootstrap.WSLauncher
     <fully_qualified_class_name_to_run> <your_application_parameters>
     ```

Chapter 41. Deploying data access resources

This page provides a starting point for finding information about data access. Various enterprise information systems (EIS) use different methods for storing data. These backend data stores might be relational databases, procedural transaction programs, or object-oriented databases.

The flexible IBM WebSphere Application Server provides several options for accessing an information system backend data store:

- Programming directly to the database through the JDBC 4.0 API, JDBC 3.0 API, or JDBC 2.0 optional package API.
- Programming to the procedural backend transaction through various J2EE Connector Architecture (JCA) 1.0 or 1.5 compliant connectors.
- Programming in the bean-managed persistence (BMP) bean or servlets indirectly accessing the backend store through either the JDBC API or JCA-compliant connectors.
- Using container-managed persistence (CMP) beans.
- Using the IBM data access beans, which also use the JDBC API, but give you a rich set of features and function that hide much of the complexity associated with accessing relational databases.

Service Data Objects (SDO) simplify the programmer experience with a universal abstraction for messages and data, whether the programmer thinks of data in terms of XML documents or Java objects. For programmers, SDOs eliminate the complexity of the underlying data access technology such as, JDBC, RMI/IIOP, JAX-RPC, and JMS, and message transport technology such as, java.io.Serializable, DOM Objects, SOAP, and JMS.

Deploying data access applications

Deploying a data access application includes more than installing your web application archive (WAR) or enterprise archive (EAR) file onto a server. Deployment can include tasks for configuring your application to use the data access resources of the server and overall runtime environment.

Before you begin

You can deploy only application code that is assembled into the appropriate modules. See the topic, Assembling data access applications for guidelines, for this process.

About this task

Perform the following steps if your application requires access to a relational database (RDB). When your application requires access to a different type of enterprise information system (EIS), such as an object-oriented database or the Customer Information Control System (CICS), consult the topics, Relational resource adapters and JCA, and Accessing data using Java EE Connector Architecture connectors.

Procedure

1. If your RDB configuration does not exist, do the following steps:
   a. Create a database to hold the data.
   b. Create tables required by your application.
      If your application uses container managed persistence (CMP) entity beans to access the data You can create the tables using the data definition language (DDL) generated from the enterprise bean configuration. For more information, see the topic, Recreating database tables from the exported table data definition language.

If your application uses bean managed persistence (BMP) entity beans, or *does not* **use entity beans**

You must use your database server interfaces to create the tables.

The Enterprise JavaBeans (EJB) to RDB Mapping wizard of an assembly tool is also used to create your database tables for either type of entity bean. Select the top-down mapping option in the wizard. However, this option does not give you direct control in naming the RDB elements or choosing column types. Additionally, because the top-down process is automatic, it might not provide mappings to reflect the precise relationships that you intend.

If you use Rational Application Developer, consult the information center about the mapping wizard. To learn about all of your assembly tool options, see the assembly tools topic in this information center.

c. Check the data source minimum required settings by vendor to see any database vendor requirements for connecting to an application server. See the topic, Data source minimum required settings, by vendor, for instructions.

2. Optional: Map your entity beans to the database tables through the meet-in-the-middle mapping option of an assembly tool. Complete this step only if you did not create your database schema through the top-down mapping option, did not generate your mapping relationships through bottom-up mapping, or did not generate mappings during the application assembly process. For information about the top-down mapping option see the information center for Rational Application Developer.
3. Install your application onto the application server. See the topic, Installing enterprise application files. When you install the application, you can alter data access settings that were made during application assembly, or, if they were omitted from the assembly process, set them for the first time. These settings include resource bindings and resource authentication aliases, which are addressed in the following substeps:
   a. Bind application resource references to the data sources, or other resource objects, that provide database connectivity. For details on the concept of binding, see the topic, Data source lookups for enterprise beans and web modules.

      Tip: After deployment, you can use the WebSphere Application Server administrative console to alter resource bindings. Click **Applications** > **Application Types** > **Webphere enterprise applications** > ***application_name***, and select the link to the appropriate mapping page. For example, if you want to alter the binding of an EJB module resource, you might click **Map data sources for all 2.x CMP beans**. For a web module resource, click **Resource references**.

   b. Define authentication alias data for resources that must be authenticated with the backend through *container-managed* authorization. In this security configuration, WebSphere Application Server performs EIS signon for data source or connection factory connections. Consult the topic, J2EE connector security for detailed reference on resource authentication.
4. Start the deployed application files using the administrative console, the wsadmin scripting tool **`startApplication`** command, or your own Java program.
5. Save the changes to your administrative configuration.
6. Test the application. For example, point a web browser at the URL for a deployed application and examine the performance of the application.

What to do next

If the application does not perform as wanted, update the application, then save and test it again.

Available resources

Use this page to select configured resources that you want to bind to the resource references of the enterprise beans or web modules in your application.

To view this administrative console page:

1. Click **Applications** > **Application Types** > **Websphere enterprise applications** > ***application_name***.
2. Click the link for any of these resource configuration pages:
   - **Resource references**
   - **Map data sources for all 2.x CMP beans**
   - **Provide default data source mapping for modules containing 2.x entity beans**
3. Locate the table row of the EJB or web module that you want to map to a different resource.
4. Within the row, locate the JNDI name of the resource that is currently bound to the EJB or web module.
5. Click **Browse**.

   You now see **Available resources**.

Each table row corresponds to a resource that you can bind to your enterprise bean or web module.

Select
Select the resource that you want to bind to the resource reference of your module.

JNDI name
The Java Naming and Directory Interface (JNDI) name of the resource that you want to bind to the resource reference of your module.

Information	Value
Data type	String

Scope
The scope of the resource. Note that this administrative console page displays only resources that are configured for a scope at which your application operates.

Description
The text description of the resource.

Map data sources for all 1.x CMP beans
Use this page to designate how the container-managed persistence (CMP) 1.x beans of an application map to data sources that are available to the application.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Map data sources for all 1.x CMP beans**.

Guidelines for using this administrative console page:

- The table depicts the 1.x CMP bean contents of your application.
- Each table row corresponds to a CMP bean within a specific EJB module. A row shows the JNDI name of the data source mapping target of the bean *only* if you bound them together during application assembly or installation. For every data source that is displayed, you see the corresponding security configuration.
- To set your mappings:
  1. Select a row. Be aware that if you check multiple rows on this page, the data source mapping target that you select in step 2 applies to all of those CMP beans.
  2. Click **Browse** to select a data source from the new page that is displayed, the Available Resources page. The Available Resources page shows all data sources that are available mapping targets for your CMP beans.
  3. Click **Apply**. The console displays the 1.x CMP bean data sources page again. In the rows that you previously selected, you now see the JNDI name of the new resource mapping target.

4. *Before* you click **OK** to save your new configuration, set the security parameters for the data source. Use the following steps.

- To specify data source security settings:
  1. Select one or more rows in the table.
  2. Type in a user name and password that comprise the authentication alias for signing on to the data source. If these entries are not listed in the application Java Platform, Enterprise Edition (Java EE) Connector (J2C) authentication data list, you must input them into the list after saving your settings on this page. Read the information center topic on managing Java EE Connector Architecture authentication data entries for more information.
  3. Click **Apply** that immediately follows the user name and password input fields.
- Repeat all of the previous steps as necessary.
- Click **OK** to save your settings.

Select
Select the check boxes of the rows that you want to edit.

EJB
The name of an enterprise bean in the application.

EJB Module
The name of the module that contains the enterprise bean.

URI
Specifies location of the module relative to the root of the application EAR file.

JNDI name
The Java Naming and Directory Interface (JNDI) name of the data source that is configured for the enterprise bean.

Information	Value
Data type	String

User name
The user name and password that comprise the authentication alias for securing the data source.

Map default data sources for modules containing 1.x entity beans
Use this page to set the default data source mapping for EJB modules that contain 1.x container-managed persistence (CMP) beans. Unless you configure individual data sources for your 1.x CMP beans, this default mapping applies to all beans within the module.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Map default data sources for modules containing 1.x entity beans**.

Guidelines for using this administrative console page:

- The page displays a table that depicts the EJB modules in your application that contain 1.x CMP beans.
- Each table row corresponds to a module. A row shows the JNDI name of the data source mapping target of the EJB module *only* if you bound them together during application assembly. For every data source that is displayed, you see the corresponding security configuration.
- To set your default data source mappings:
  1. Select a row. Be aware that if you check multiple rows on this page, the data source mapping target that you select in step 2 applies to all of those EJB modules.

2. Click **Browse** to select a data source from the new page that is displayed, the Available Resources page. The Available Resources page shows all data sources that are available mapping targets for your EJB modules.
3. Click **Apply**. The console displays the 1.x entity bean data sources page again. In the rows that you previously selected, you now see the JNDI name of the new resource mapping target.
4. *Before* you click **OK** to save your new configuration, set the security parameters for the data source. Use the following steps.

- To specify security settings for the default data source:
  1. Select a row. Be aware that if you check multiple rows on this page, the security settings that you select later apply to all of those data sources.
  2. Type in a user name and password that comprise the authentication alias for signing on to the data source. If these entries are not listed in the application Java Platform, Enterprise Edition (Java EE) Connector (J2C) authentication data list, you must input them into the list after saving your settings on this page. Read the information center topic on managing Java EE Connector Architecture authentication data entries for more information.
  3. Click **Apply** that immediately follows the user name and password input fields.
- Repeat all of the previous steps as necessary.
- Click **OK** to save your work.

Select
Select the check boxes of the rows that you want to edit.

EJB Module
The name of the module that contains the 1.x enterprise beans.

URI
Specifies location of the module relative to the root of the application EAR file.

JNDI name
The Java Naming and Directory Interface (JNDI) name of the default data source for the EJB module.

Information	Value
Data type	String

User name
The user name and password that comprise the authentication alias for securing the data source.

Map data sources for all 2.x CMP beans settings
Use this page to map container-managed persistence (CMP) 2.x beans of an application to data sources that are available to the application.

To view this administrative console page, click **Applications** > **Application Types** > **Websphere enterprise applications** > ***application_name*** > **Map data sources for all 2.x CMP beans**.

Each table row corresponds to a CMP bean within a specific EJB module. A row shows the JNDI name of the data source mapping target of the bean only if you bound them together during application assembly. For every data source that is displayed, you see the corresponding security configuration.

Set Multiple JNDI names
Specify the Java Naming and Directory Interface (JNDI) name for multiple EJB modules. Select one or more EJB modules from the table, and select a JNDI name from this list to configure the EJB modules with that JNDI name.

Information	Value
Data type	Drop-down list

Set Authorization Type

Specify the authorization type for securing the data source. Select one or more EJB modules from the table to set the authorization type.

Select either **Container** or **Application** from the displayed list. Container-managed authorization indicates that WebSphere Application Server performs signon to the data source. Application-managed authorization indicates that the enterprise bean code performs signon.

Modify Resource Authentication Method

Specify the authorization type and the authentication method for securing the data source. Select one or more EJB modules from the table to modify the resource authentication method.

You can choose between the following authentication methods:

- **None**:
  1. Determine which data source configurations to designate with no authentication method.
  2. Select the appropriate table rows.
  3. Select **None** from the list of authentication method options that precede the table.
  4. Click **Apply**.
- **Use default method (many-to-one mapping)**:
  1. Determine which data source configurations to designate with the WebSphere Application Server DefaultPrincipalMapping login configuration. Apply this option to each data source individually if you want to designate different authentication data aliases. See the information center topic on J2EE Connector security for more information on the default mapping configuration.
  2. Select the appropriate table rows.
  3. Select **Use default method (many-to-one mapping)** from the list of authentication method options that precede the table.
  4. Select an authentication data entry or alias from the list.
  5. Click **Apply**.
- **Use Kerberos authentication**: Specifies to use the Kerberos authentication method.
  1. Ensure that you have configured the Kerberos authentication mechanism in the application server.
  2. Select the appropriate table row.
  3. Select **Use Kerberos authentication** from the list of authentication method options that precede the table.
  4. Select an application login configuration from the list.
  5. Click **Apply**.
  6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

  The application server will attempt to verify that you are connecting to the correct type of database when you select this option.
- **Use trusted connections (one-to-one mapping)**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Ensure that the database to which the modules will connect is configured for trusted connections.
  4. Select **Use trusted connections (one-to-one mapping)** from the list of authentication method options that precede the table.

5. Select an application login configuration from the list.
6. Click **Apply**.

The application server will attempt to verify that you are connecting to the correct type of database when you select this option.

- **Custom login configuration**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Select **Use custom login configuration** from the list of authentication method options that precede the table.
  4. Select an application login configuration from the list.
  5. Click **Apply**.
  6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

Select
Select the check boxes of the rows that you want to edit.

EJB
The name of an enterprise bean in the application.

EJB Module
The name of the module that contains the enterprise bean.

URI
Specifies location of the module relative to the root of the application EAR file.

Target resource JNDI name
Specifies the resource to which the CMP bean is bound.

Resource authorization
Specifies the current setting for the resource authorization type.

Modify this setting with **Set authorization type**.

Map data sources for all 2.x CMP beans
Use this page to set the default data source mapping for EJB modules that contain 2.x container-managed persistence (CMP) beans. Unless you configure individual data sources for your 2.x CMP beans, this default mapping applies to all beans within the module.

To view this administrative console panel, click **Applications** > **Application Types** > **Websphere enterprise applications** > ***application_name*** > **Map data sources for all 2.x CMP beans** .

This panel displays a table that depicts the EJB modules in your application that contain 2.x CMP beans. Each table row corresponds to a module. A row shows the JNDI name of the data source mapping target of the EJB module only if you bound them together during application assembly. For every data source that is displayed, you see the corresponding security configuration.

Set Multiple JNDI Names
Specifies the JNDI name to bind to one or more modules. Select one or more modules, click **Set Multiple JNDI Names**, and select the JNDI name for the resource to which you would like to bind the module.

Set Authorization Type

Specifies the authorization type that you to use for the modules. Select one or more modules, click **Set Authorization Type**, and select the authorization type.

You can choose:

- Per application - indicates that the enterprise bean code performs signon.
- Container - indicates that the application server performs signon to the data source.

Modify Resource Authentication Method

Specifies the resource authentication method for the modules that you have configured with container-managed authorization. Select one or more modules, click **Modify Resource Authentication Method**, and select the authentication method.

You can choose between the following authentication methods:

- **None**:
  1. Determine which data source configurations to designate with no authentication method.
  2. Select the appropriate table rows.
  3. Select **None** from the list of authentication method options that precede the table.
  4. Click **Apply**.
- **Use default method (many-to-one mapping)**:
  1. Determine which data source configurations to designate with the WebSphere Application Server DefaultPrincipalMapping login configuration. Apply this option to each data source individually if you want to designate different authentication data aliases. See the information center topic on J2EE Connector security for more information on the default mapping configuration.
  2. Select the appropriate table rows.
  3. Select **Use default method (many-to-one mapping)** from the list of authentication method options that precede the table.
  4. Select an authentication data entry or alias from the list.
  5. Click **Apply**.
- **Use Kerberos authentication**: Specifies to use the Kerberos authentication method.
  1. Ensure that you have configured the Kerberos authentication mechanism in the application server.
  2. Select the appropriate table row.
  3. Select **Use Kerberos authentication** from the list of authentication method options that precede the table.
  4. Select an application login configuration from the list.
  5. Click **Apply**.
  6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

  The application server will attempt to verify that you are connecting to the correct type of database when you select this option.
- **Use trusted connections (one-to-one mapping)**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Ensure that the database to which the modules will connect is configured for trusted connections.
  4. Select **Use trusted connections (one-to-one mapping)** from the list of authentication method options that precede the table.
  5. Select an application login configuration from the list.
  6. Click **Apply**.

The application server will attempt to verify that you are connecting to the correct type of database when you select this option.

- **Custom login configuration**:
  1. Determine which data source configurations to designate with a custom Java Authentication and Authorization Service (JAAS) login configuration. See the information center topic on J2EE Connector security for more information on custom JAAS login configurations.
  2. Select the appropriate table row.
  3. Select **Use custom login configuration** from the list of authentication method options that precede the table.
  4. Select an application login configuration from the list.
  5. Click **Apply**.
  6. To edit the properties of the custom login configuration, click **Mapping Properties** in the table cell.

Select
Select the check boxes of the rows you want to edit.

EJB Module
Specifies the name of the module that contains the 2.x enterprise beans.

URI
Specifies location of the module relative to the root of the application EAR file.

JNDI name
Specifies the Java Naming and Directory Interface (JNDI) name of the default data source for the EJB module.

Information	Value
Data type	String

Resource authorization
Specifies the authorization type and the authentication method for securing the data source.

Extended Datasource Properties
When selected, you will be directed to a panel on which you can specify extended properties that the module can use for the DB2 data source.

The application server will attempt to verify that you are connecting to the correct type of database when you select this option.

Installing a resource adapter archive

The application server uses the classes and other code that comprise a resource adapter archive (RAR) to support the resource adapters that you configure.

Before you begin

A RAR file, which is often called a Java EE Connector Architecture (JCA) connector, must comply with the JCA Specification. You can meet these requirements by using a supported assembly tool to assemble a collection of Java archive (JAR) files, other runnable components, and utility classes into a deployable resource adapter archive (RAR). You can then install the RAR file in the application server.

When you are using optimized local adapters, set up your server environment in the daemon group before installing the `ola.rar` file in the application server. The `ola.rar` file is located in the `WAS_HOME/installableApps` directory.

You can also use the wsadmin scripting file, olaRar.py, to install the RAR file.

To read about setting up your server environment, see the topic, Enabling the server environment to use optimized local adapters. You can read about the `olaRar.py` script in the topic, olaRar.py scripting file.

About this task

A resource adapter archive provides the classes and other code to support a resource adapter for access to a specific EIS, such as the Customer Information Control System (CICS). Therefore, you can only configure resource adapters for an EIS after you install the appropriate RAR file.

Important: When you use the **Install RAR** dialog to install a RAR file, the scope you define on the Resource Adapters page has no effect on where the RAR file is installed. You can install RAR files only at the node level, which you specify on the Install RAR page. To set the scope of an RAR file to a specific cluster, or server, after you install the RAR file at each node level, create a copy of the RAR file with the appropriate cluster or server scope.

Procedure

1. Navigate to the **Resource adapter** panel. Click **Resources** > **Resource Adapters** > **Resource adapters**.
2. Install a new resource adapter archive.
   a. Click **Install RAR**. A dialog opens for installing a RAR file and configuring the associated resource adapter. Only click **New** if you want to configure a new resource adapter for a previously installed RAR file.
   b. Browse to find the appropriate RAR file.
      - If your RAR file is located on your local workstation, select **Local path**, and browse to find the file.
      - If your RAR file is located on your server, select **Remote file system**, and specify the fully qualified path to the file.
   c. Click **Next**.
3. Configure the resource adapter name and any other properties needed under *General Properties*. For more details on the settings that you can configure, such as the J2C connection factories, see the topics Installing resource adapters within applications and Configuring resource adapters.
4. Click **OK**.
5. Optional: Create a copy of the RAR file with a different scope level. After you install the RAR file at each node level, you can create another copy of the file that has a specific server or cluster as the scope for that file.

   Note:
   - If you do not create a copy of your RAR at the cluster scope, then you must create identical factories (connection factories, admin object, and activation specifications) at the node level for each of your nodes in the cluster. By creating the copy of your RAR, you provide a placeholder for your factories and circumvent the need to create identical factories at the node level for each of your nodes in the cluster.
   - You must still install the RAR binaries (files, such as jars and xml deployment files) on each node for the RAR to operate successfully.

   a. Click **Resources**.
   b. Click **Resource Adapters**.
   c. Select the scope level and then click **NEW**.
   d. Choose the RAR file from the installed archive path.
   e. Click **OK**.

Results

You have installed a resource adapter archive that provide access to the EIS when it is properly configured. If you must configure more settings, or change some settings that were configured during the installation process, refer to the topic on configuring a resource adapter in the administrative console for more information.

Installing resource adapters embedded within applications

Install resource adapters in your applications so they can access outside data sources.

Before you begin

The JCA Version 1.6 specification adds support for Java annotations in RAR modules. For more information on annotation support see the topic, JCA 1.6 support for annotations in RAR modules.

About this task

Procedure

1. Assemble an application with RAR modules in it. See the topic Assembling applications for more information.
2. Install the application. Follow the steps in the topic Installing a new application.

   In the **Map modules to servers** step, specify target servers or clusters for each RAR file. Be sure to map all other modules that use the resource adapters defined in the RAR modules to the same targets. Also, specify the web servers as targets that serve as routers for requests to this application. The plug-in configuration file (`plugin-cfg.xml`) for each web server is generated based on the applications that are routed through it.

   In the **Metadata for modules** step of installing an application, you can set or unset the metadata-complete flag as discussed in the topic, JCA 1.6 support for annotations in RAR modules.

   Note: When installing a RAR file on a server, the application server looks for the manifest (MANIFEST.MF) for the connector module. The application server first looks for the RAR file's `connectorModule.jar` file and loads the manifest from the `connectorModule.jar` file. If the class path entry is in the manifest from the `connectorModule.jar`file, the RAR uses that class path.

   To ensure that the installed connector module finds the classes and resources that it needs, check the Class path setting for the RAR using the administrative console. For more information on how to check this setting, see the topics Resource adapter settings and WebSphere relational resource adapter settings.
3. Click **Finish** > **Save** to save the changes.
4. Create connection factories for the newly installed application.

   See the topic, Configuring connection factories for resource adapters within applications to view the steps to complete this step.

Results

Note: A given native library can only be loaded one time for each instance of the Java virtual machine (JVM). Because each application has its own class loader, separate applications with embedded RAR files cannot both use the same native library. The second application receives an exception when it tries to load the library.

If any application deployed on the application server uses an embedded RAR file that includes native path elements, then you must always ensure that you shut down the application server cleanly, with no outstanding transactions. If the application server does not shut down cleanly it performs *recovery* upon server restart and loads any required RAR files and native libraries. On

completion of recovery, do not attempt any application-related work. Shut down the server and restart it. No further recovery is attempted by the application server on this restart, and normal application processing can proceed.

Install RAR

Use this page to install a resource archive (RAR) file in one of two ways. You can either upload a RAR file from the local file system, or specify an existing RAR file on a server. The RAR file must be installed at the node level, and you can select the node on this page.

To view this page in the administrative console click **Resources** > **Resource Adapters** > **Resource Adapters** > **Install RAR**.

For information about installing a resource adapter, see the topic, Installing a resource adapter archive (RAR) file.

Scope

Specifies the scope of the resource adapter. Only applications that are installed within this scope can use this adapter.

Local file system

Specifies the path of a RAR that resides on the same server as the console.

Information	Value
Data type	String

Remote file system

Specifies the path of a RAR that resides on one of the nodes of the cell.

Information	Value
Data type	String

Deploying SQLJ applications

Use Structured Query Language in Java (SQLJ) to develop data access applications that connect to DB2 databases. SQLJ is a set of programming extensions that enable you to use the Java programming language to embed statements that provide SQL (Structured Query Language) database requests.

About this task

The advantages of developing applications with SQLJ include improved performance and a shorter, more efficient development cycle. You can achieve the following with SQL:

- Improve performance by using static SQL statements.
- Reduce the development cycle:
  - Write less code with the simpler SQLJ syntax, which reduces the number of lines of code that is required to execute statements, set parameters, and retrieve parameters.
  - Detect programming errors earlier in the development phase with the online check function, which performs data type validation and schema validation. See the DB2 documentation for a complete list of customization options.

Consider using SQLJ in situations where dynamic SQL is not needed, and where applications use DB2 as the database server.

The application server includes enhanced SQLJ support for applications that use container-managed persistence (CMP). The enhanced support includes the following items: include:

- Deploying CMP beans during the application installation in the application server.

- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without needing to reinstall the application.

These enhancements reduce the complexity of installing, deploying, and customizing SQLJ applications for both container-managed and bean-managed persistence.

Procedure

1. Acquire the required drivers to deploy an SQLJ application in the application server. You need the following files, depending on the JDBC provider that you use:

JDBC provider type	Required files
DB2 Using IBM JCC Driver This driver is also known as: • IBM Data Server Driver for JDBC and SQLJ • IBM DB2 Driver for JDBC and SQLJ • IBM DB2 Universal JDBC Driver.	db2jcc.jar or db2jcc4.jar
DB2 Universal JDBC driver (deprecated)	db2jcc.jar

2. Deploy the SQLJ application.
   - Deploy applications that use container-managed persistence (CMP):
     - "Deploying SQLJ applications that use container-managed persistence (CMP)" with the DB2 Using IBM JCC Driver.
     - "Deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool" on page 2054.
   - "Deploying SQLJ applications that use bean-managed persistence, servlets, or sessions beans" on page 2056.
   - "Using embedded SQLJ with the DB2 for z/OS Legacy driver" on page 2066 (deprecated).
3. Customize and bind the SQLJ profiles. Before the application server can use an SQLJ application, the SQLJ statements must be processed for the database server. By default, four DB2 packages are created in the database; one package is created for each isolation level. The customization process augments the profiles with information that is specific to the database. If you do not customize the SQLJ profiles, the SQLJ application uses dynamic SQL like a JDBC application.
   - "Customizing and binding profiles for Structured Query Language in Java (SQLJ) applications" on page 2057.
   - Customize and bind SQLJ profiles with the wsadmin scripting tool. See the topic, Customizing and binding SQLJ profiles with the wsadmin tool.
   - "Customizing and binding SQLJ profiles with the db2sqljcustomize tool" on page 2060.

Deploying SQLJ applications that use container-managed persistence (CMP)

Embed Structured Query Language in Java (SQLJ) statements in your applications to maximize the efficiency of transactions with your databases. Before your applications can take advantage of SQLJ, you must deploy the application and customize the SQLJ profiles that are created. The application server provides functionality to use SQLJ as the persistence mechanism for enterprise beans that use container-managed persistence. Deploy the CMP beans in the application server to enable SQLJ support.

Before you begin

You need an application that uses SQLJ and container-managed persistence. Develop this application in Rational Application Developer or another development tool.

About this task

Deploy SQLJ applications in the application server to simplify the process of SQLJ translation and bean deployment. The application server includes these new features for SQLJ support:

- Deploying CMP beans during the application installation in the application server.
- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without needing to reinstall the application.

You can also deploy the SQLJ application using the ejbdeploytool. Read the topic on deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool for more information.

Procedure

1. Create a top-down mapping to a DB2 database.
2. From your DB2 installation, copy the `sqlj.zip` file to a directory on your workstation.
3. Deploy the EAR file in the administrative console.
   a. Click **Applications** > **Install New application**.
   b. Select **Local file system** or **Remote file system**, and browse to the EAR file.
   c. Select **Detailed - Show all installation options and parameters**. Click **Next**.
   d. In **Step 1: Select installation options**, select **Deploy enterprise beans**. Configure any other options, and click **Next**.
   e. In **Step 3: Provide options to perform the EJB deploy**, select **SQLJ** for **Deploy EJB option - Database access type**.
   f. Enter the location of the `sqlj.zip` file in the **SQLj class path** field.
   g. Complete the installation process for the application.

What to do next

After the enterprise application is deployed, customize the SQLJ profiles using the administrative console, scripting, or the db2sqljcustomize tool:

- For administrative console support, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
- For scripting support, read the topic on the application management command group for the AdminTask object.
- For use of the db2sqljcustomize tool, read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool.

Deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool

Embed Structured Query Language in Java (SQLJ) statements in your applications to maximize the efficiency of transactions with your databases. Before your applications can take advantage of SQLJ, you must deploy the application and customize the SQLJ profiles that are created. The application server provides functionality to use SQLJ as the persistence mechanism for enterprise beans that use container-managed persistence. Use the ejbdeploy tool to deploy the application.

About this task

You can deploy SQLJ applications with the ejbdeploy tool to deploy the enterprise application in a stand-alone environment.

Alternatively, the application server includes enhanced SQLJ support for applications that use container-managed persistence (CMP). The new features include:

- Deploying CMP beans during the application installation in the application server.

- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without needing to reinstall the application.

These enhancements reduce the complexity of installing, deploying, and customizing SQLJ applications for both container-managed and bean-managed persistence.Read the topic on deploying SQLJ applications that use container-managed persistence (CMP) for more information.

Procedure

1. Create a top-down mapping to a DB2 database.
2. From your DB2 installation, copy the `sqlj.zip` file to a directory on your workstation.
3. Modify the Java build path of your enterprise bean JAR project to include the `sqlj.zip` file.
4. Use Rational Application Developer or the DB2 SQLJ translator to automatically translate SQLJ.
   - Use Rational Application Developer:
     a. From the Project Navigator, click ***EJB_JAR_PROJECT_NAME*** > ***SOURCE_FOLDER*** > **META-INF** > **backends** > ***database_version***.
     b. Open Map.mapxmi in the Mapping editor.
     c. On the **Overview** panel, highlight the name of your JAR project in the Enterprise Beans column. You must highlight the name of the JAR project, not the name of one of the enterprise beans that is listed.
     d. On the **Properties** panel, expand **SQLJ**.
     e. Set **Is using SQLJ?** to `True`.
     f. Set **Translator Module** to the fully qualified path of the sqlj.zip file on your workstation.
     g. Save the Map.mapxmi file.
     h. Export the enterprise archive (EAR) file.
   - Use the DB2 SQLJ translator. This tool creates a .java version of your .sqlj file and a serialized profile, with a .ser extension, that is used later in processing. Refer to the DB2 documentation for more information on the SQLJ translator tool.
5. Deploy the EAR file with the ejbdeploy tool.
   a. Verify that the app_server_root/bin directory is in your class path.
   b. Run the ejbdeploy command utility with the `-sqlj` option. The ejbdeploy command will generate an EAR file with the name you specify and an Ant script with the name *application_name*.ear.xml.

      For example: :

      ```
      ejbdeploy d:\application_name.ear
                working d:\deployed_application_name.ear
                -sqlj
                -dbvendor DB2UDB_V81
                -cp "C:\PROGRA~1\IBM\SQLLIB\java\sqlj.zip"
      ```

      Note: Supply the location of the SQLJ translator sqlj.zip file with -cp, which is the class path option The ejbdeploy command does not access `sqlj.zip` from your system class path.
6. Choose the option for customization.
   - Use the application server's SQLJ support. Install the deployed application to customize the SQLJ profiles with the application server or scripting.
     a. Install the enterprise application in the application server.

        Note: Do not select **Deploy enterprise beans** during the application installation process in the administrative console. If you redeploy the enterprise beans from the administrative console, you will lose the customization changes that you have made.
     b. Customize the SQLJ profiles.
        – For administrative console support, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.

  - For scripting support, read the topic on the application management command group for the AdminTask object.
- Customize and bind the SQLJ profiles with the db2sqljcustomize tool. Read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool.

Deploying SQLJ applications that use bean-managed persistence, servlets, or sessions beans

You can embed Structured Query Language in Java (SQLJ) statements in your applications to maximize the efficiency of transactions with your databases. Before your applications can take advantage of SQLJ, deploy the application and customize the created SQLJ profiles. You can use Rational Application Developer or the DB2 SQLJ translator to translate the application before deploying it on the application server.

Before you begin

Create an SQLJ application using Rational Application Developer or another development tool.

About this task

To deploy SQLJ applications that do not use container-managed persistence, translate the SQLJ application first to configure it for the application server environment. After translation, customize the SQLJ profiles in the application server, with scripting, or with the db2sqljcustomizer tool.

SQLJ support for applications that use bean-managed persistence include these features:

- Customizing and binding SQLJ profiles with the administrative console or scripting.
- Customizing and binding SQLJ applications again without reinstalling the application.

Procedure

1. Optional: Create a backup copy of your .java file. For example if your file is called `MyServlet.java`, copy `MyServlet.java` to `MyServlet.java.bkup`.
2. Optional: Rename your .java file to a file name with an `.sqlj` extension. For example, if your application is a servlet named `MyServlet.java`, rename `MyServlet.java` to `MyServlet.sqlj`
3. Optional: Edit the SQLJ file to convert the JDBC syntax to SQLJ syntax. When using SQLJ, if you want connection management for the application server to function properly, specify correct connection contexts.

   For example, convert the following JDBC operation:

   ```
   Connection con = dataSource.getConnection();
   Statement stmt = con.createStatement();
   stmt.execute("INSERT INTO users VALUES (1, 'user1')");
   con.commit();
   ```

   to the following SQLJ:

   ```
   // At the top of the file and just below the import statements, define Connection_Context
   #sql context Connection_context;
   .
   .
   Connection con = dataSource.getConnection();
   .
   .
   Connection_context ctx1 = new Connection_context(con);
   .
   .
   #sql [ctx1] {INSERT INTO users VALUES (1, 'user1')};
   .
   .
   con.commit(); ctx1.close();
   ```

When you run the SQLJ translator, the `.java` file that is created has the same name as your old `.java` file. This provides you with a seamless transition to the SQLJ technology.

4. From your DB2 installation, copy the `sqlj.zip` file to a directory on your workstation. Modify the Java build path of your enterprise bean Java archive (JAR) file project to include the `sqlj.zip` file.
5. Use Rational Application Developer or the DB2 SQLJ translator to automatically translate SQLJ.
   - Use Rational Application Developer:
     a. In the Project Navigator, right-click your JAR project, and select **Add SQLJ Support...**.
     b. Select the check boxes for the applications for which you want SQLJ support.
     c. In the **SQLJ JAR file** field, type the fully qualified path to the `sqlj.zip` file that you previously copied to your workstation.
     d. Click **Finish**.
     e. Export the enterprise archive (EAR) file.
   - Use the DB2 SQLJ translator. This tool creates a `.java` version of the `.sqlj` file and a serialized profile, with an `.ser` extension, that is used later in processing. Refer to the DB2 documentation for more information about the SQLJ translator tool.
6. Package your JAR file for the enterprise application.
7. Install the application onto the application server, or customize the profiles with the db2sqljcustomize tool.
   - Customize the profiles with the application server.
     a. Package the JAR file for your enterprise beans, servlets, and any `.ser` files into an enterprise archive.
     b. Install the application in the application server, and customize SQLJ profiles with the administrative console or the wsadmin tool.

        Note: Do not select **Deploy enterprise beans** during the application installation process in the administrative console. If you redeploy the enterprise beans from the administrative console, you lose the customization changes that you have made.

        The application server provides enhanced support for SQLJ applications. Install the SQLJ application in the application server, and you can customize and bind SQLJ profiles through the administrative console or scripting:
        – To customize the SQLJ profiles with the administrative console, read the topic about customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
        – To customize SQLJ profiles with scripting, read the topic about the application management command group for the AdminTask object.
   - To use the db2sqljcustomize tool, read the topic about customizing and binding SQLJ profiles with the db2sqljcustomize tool for more information.

Customizing and binding profiles for Structured Query Language in Java (SQLJ) applications

Simplify the process of customizing and binding SQLJ profiles for your applications by performing these functions in the administrative console or with scripting. SQLJ profiles must be customized and bound before the enterprise application can use the application's embedded SQL.

Before you begin

You must have an SQLJ application that has already been deployed and installed in the application server.

For SQLJ applications that use container-managed persistence, you can deploy the application in two ways:

- Deploy the SQLJ application in the application server. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) for more information.

- Deploy SQLJ applications with the ejbdeploy tool. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool.

For SQLJ application that use bean-managed persistence, see the topic on deploying SQLJ applications that use bean-managed persistence, servlets, or session beans.

About this task

To take advantage of SQLJ applications in the application server, you need to customizing the SQLJ profiles that contain the embedded SQL statements. By default, four DB2 packages are created in the database; one for each isolation level. The customization process augments the profiles with information that is specific to the DB2 database. The database uses this information at run time.

In addition to profile customization, you need to bind the customized profiles to the DB2 database. Profile binding should only take place after the SQLJ profiles are customized.

You can also customize and bind profiles with scripting or the db2sqljcustomize tool:

- For scripting support, read the topic on the application management command group for the AdminTask object.
- For information on the db2sqljcustomize tool, read the topic on customizing and binding SQLJ profiles with the db2sqljcustomize tool for more information. If you customize profiles with the db2sqljcustomize tool, you will need to reinstall the application.

Procedure

1. Make sure the necessary database tables exist, as described in the topic on deploying data access applications.
2. Navigate to the SQLJ application that is installed in the application server. Click **Applications** > **Websphere enterprise applications** > ***app_name***.

   Note: Do not run multiple sessions of the administrative console to customize and bind profiles that are in the same EAR file.
3. Navigate to the SQLJ profiles section. Click **SQLj profiles**. When you click this link, the application server expands the EAR file for the application into a temporary directory; there might be a delay before the panel for SQLJ profiles is displayed.
4. Select **Customize and bind profiles** or **Bind packages**. Choose your option based on the profiles with which you are working:
   - If your profiles have not been customized, or you want to customize the profiles again, choose **Customize and bind profiles**.
   - If the profiles are already customized, choose **Bind packages**.
5. Choose to select profiles or a profile group to customize and bind.
   - Select profiles from the list that is provided.
     a. Select the profiles from the list and click **Add**. The list displays the SQLJ profiles that are present in the enterprise application.

        Note:
        – Select more than one profile by holding CTRL.
        – Select a contiguous list of profiles by selecting the first profile name, holding SHIFT, and selecting the last profile. You will select the first profile, last profile, and any profiles in the middle.

     b. Select **Customize/bind the selected SQLj profiles as a group** This option specifies that the application server will create a .grp file that contains the SQLj profiles that are processed. You can use the .grp file for other binding operations in the future. After you have completed this panel and click **OK**, you will be given an option to download the .grp file.

- Select **Use a profile group file to specify profiles to customize/bind**. Select this to specify a profile group to process. Click **Browse...** to locate the file on the system.

6. Complete the necessary information to connect to the database. You need to complete the following fields:

   Database URL
   Specifies the URL of the database to which the profile/s will be bound. The typical syntax is:
   `jdbc:db2://<host name="">:<port>/<database name="">.</database></port></host> or`

   or
   fully_qualified_host_name:*port*

   User Specifies the user ID for the database administrator on the server where the database is located.

   Password
   Specifies the password for the database administrator on the server where the database is located.

   Additional options
   Specifies additional options to use during the customization and bind processes. See the DB2 documentation for a complete list of customization options.

   Class path
   Specifies the class path where sqlj.zip, and db2jcc.jar or db2jcc4.jar are located.

7. Click **OK**.

   Note: If you are processing a large enterprise application, or you are processing many SQLJ profiles, the process might take longer than the default timeout for the administrative console. The default connection timeout for the application server's administrative console is set to 30 minutes. If the default timeout is reached and you lose the connection to the server, you can check the system output log for the final results of the customization and bind process.

   To prevent this disconnection, configure the console session timeout to a longer period of time. After a successful customization and binding process, check the system output log for the total processing time. Use that time period as a basis for the new timeout value. For information about how to configure the console timeout, see the topic on changing the console session expiration.

Results

After the application server finishes processing the SQLJ profiles, you will see the results from the customization and binding. The results panel displays messages from the database server, as well as summary results from the application server.

If the operation completed successfully, the following message will be printed to the system log:

```
ADMA0507I=ADMA0507I: The SQLJ operation on application {0} completed successfully.  Exit code: {1}
ADMA0507I.explanation=This informational message indicates the program status.
ADMA0507I.useraction=No user action is required.
```

If the operation did not complete successfully, the following message will be printed to the system out log:

```
ADMA0506I=ADMA0506I: The SQLJ operation on application {0} did not complete successfully.  Exit code: {1}
ADMA0506I.explanation=The SQLJ operation encountered a problem.  This informational message indicates
the program status. Prior messages in the command output give details of the problem.
ADMA0506I.useraction=Check the command output for the cause of the problem.
```

Customizing and binding SQLJ profiles with the db2sqljcustomize tool

Customize and bind SQLJ profiles with the db2sqljcustomize tool before you install the SQLJ application in the application server.

Before you begin

To perform this task, you must have SQLJ application that has been deployed, but the application should not be installed in the application server. If the application is already installed in the application server, you will need to reinstall the application after you customize the profiles. You also need serialized profiles for the SQLJ application.

For SQLJ applications that use container-managed persistence, you can deploy the application in two ways:

- Deploy the SQLJ application in the application server. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) for more information.
- Deploy SQLJ applications with the ejbdeploy tool. See the topic on deploying SQLJ applications that use container-managed persistence (CMP) with the ejbdeploy tool.

For SQLJ application that use bean-managed persistence, see the topic on deploying SQLJ applications that use bean-managed persistence, servlets, or sessions beans.

About this task

To take advantage of SQLJ applications in the application server, you need to customize the SQLJ profiles. The customization process augments the profiles with information that is specific to the DB2 database. The database uses this information at run time. By default, four DB2 packages are created in the database; one package is created for each isolation level.

The application server supports customizing and binding the SQLJ profiles in the administrative console or with scripting:

- For administrative console support, read the topic on customizing and binding profiles for Structured Query Language in Java (SQLJ) applications.
- For scripting support, see the topic on the application management command group for the AdminTask object.

Procedure

1. Make sure the necessary database tables exist, as described in the topic on deploying data access applications.
2. Transfer the serialized profiles to the environment on which you installed your application. Alternatively, use the Java `jar` command to extract the serialized profiles from the JAR file in your installed EAR directory.
3. Add the location for the SQLJ profiles and the application's JAR file to your environment's class path.
4. Make sure the necessary database tables exist, as described in the topic on deploying data access applications.
5. Optional: If your application is not running in a clustered environment, you can use the Ant script to make customization easier. If you run a batch SQLJ customization against an EAR file with the ejbdeploy tool, the tool produces an Ant script that is named *application_name*.ear.xml. You can use this script file to run the DB2 customizer program against the serialized profiles in all of the enterprise bean JAR files for the associated EAR file. The script updates each enterprise bean's JAR file with a serialized profile and replaces the JAR files in the existing EAR file with the modified versions.
   a. Change the values of the database URL, and the database user and password properties in ejbdeploy.sqlj.properties. This file is a common file to all Ant scripts that are generated by the ejbdeploy command. The ejbdeploy.sqlj.properties script defines the global properties for:
      - Database URL - db.url

- User - db.user
- Password - db.password

The Ant script uses the URL, user, and password properties in the serialized profile to customize the profile. By default, the properties for the serialized profile are created from the global properties.

b. Run the Ant script, specifying the `properties` target. For example:

```
ws_ant -buildfile application_name.ear.xml properties
```

This script creates the properties file, *application_name*.ear.properties. The *application_name*.ear.properties file contains properties that specify the default names for the packages corresponding to each serialized profile in the EAR file. This is a sample properties file:

```
url.MyEJB1.jar.DB2UDBNT_V8_1=jdbc:db2://localhost:50000/MyDB1
user.MyEJB1.jar.DB2UDBNT_V8_1=dbuser
password.MyEJB1.jar.DB2UDBNT_V8_1=dbpassword
pkg.MyEJB1.jar.DB2UDBNT_V8_1=TEST
url.MyEJB2.jar.DB2UDBNT_V8_1=jdbc:db2://localhost:50000/MyDB2
user.MyEJB2.jar.DB2UDBNT_V8_1=dbuser
password.MyEJB2.jar.DB2UDBNT_V8_1=dbpassword
pkg.MyEJB2.jar.DB2UDBNT_V8_1=WORK
```

c. Use the DB2 Control Center to identify the packages that are installed in the database. The DB2 SQLJ customizer requires a type 4 database URL in the form of:

```
jdbc:db2://host-name:port/database-name
```

It also requires a user and password. The value of the port is 50000, unless you change it when you install DB2.

d. Change the names that are used by the script file to ensure that the names for each customization profile do not conflict with existing package names that are in the database. Ant scripts that are generated for different EAR files use the same package names by default, and the script will overwrite existing packages unless you change the names. Overwritten packages can cause errors at run time.

DB2 uses the first seven characters of the package name. The DB2 customizer uses this name to create four packages in the database. For example, if you specify the name TEST, the DB2 customizer will create packages called TEST1, TEST2, TEST3, and TEST4.

e. Run the Ant script. The Ant script updates the original EAR file with the modified serialized profiles.

Note: Verify that you have db2jcc.jar in the class path. This file should have been added to the class path environment variable when DB2 V8 FixPak1 was installed.

A sample Ant command looks like this:

```
ws_ant -Dwork.dir=tmp
       -Dscript.property.file=other.properties
       -buildfile application_name.ear.xml
```

where:

- `-buildfile` specifies the XML file to create.
- `-Dscript.property.file` specifies a different properties file. This parameter is optional. If you want your Ant script to use a another file instead of *application_name*.ear.properties, specify the Dscript.property.file property when you run the script.
- `-Dwork.dir` specifies a temporary working directory for the script. The script will create and delete files and subdirectories in this directory. If the working directory contains existing files and directories with the same name as the files and directories used by the script, the script will erase or overwrite the files and directories. This script creates and uses a directory called `tmp` as its working directory.

f. Proceed to installing the application in the application server..

6. Run the db2sqljcustomize tool to customize the SQLJ profiles that correspond to each enterprise bean's JAR file. When you generate your deployment code, serialized profiles (files with a .ser extension) that are specific to your application are created. These profiles exist in the same directory as your SQLJ files, and the files must be customized to the environment before they can be used. When you run the DB2 SQLJ customizer against the serialized profiles, you create static SQL in the database that DB2 will use at run time. The customization phase creates four database packages that contain static SQL, one for each isolation level.
   a. Optional: Consider using the SQLJ customizer tool to enable context caching for your application's data source connections. DB2 V8.1 fix pack 6 provides the new caching option with the db2sqljcustomize tool called db2optimize. You can run this option if your application uses the explicit connection context instead of the default context.

      Note:

      - SQLJ context caching support requires the DB2 with IBM JCC driver or Version 2.2 or later of the DB2 Universal JDBC Driver with APAR PQ87786 applied.
      - If you want to enable context caching for an application or BMP bean that caches connections across transaction boundaries, you cannot use shareable connections. Use the get/use/close pattern of connection usage when you invoke the db2optimize option, or an object closed exception occurs. The following code gives an example of incorrect connection usage for context caching:

        ```
        utx.begin();
             cons =ds.getConnection(
             request.getParameter("db.user"),
             request.getParameter("db.password"));
                 cmctx1 = new CM_context(cons);
                 #sql [cmctx1] {DELETE FROM cmtest WHERE id=1};
         utx.commit();
               //The next statement verifies the result:
                 #sql [cmctx1] cursor1 = {SELECT id, name FROM cmtest WHERE id=1};
        ```

        In this case, the `Select` statement elicits an object closed exception. To prevent the exception from occurring, close the connection before committing the transaction. Then get a new connection and a new context before running the `Select` statement.

      The following example code demonstrates proper syntax for running the option on the serialized profile:

      ```
      sqlj -db2optimize SQLJTransactionTest.sqlj
      db2sqljcustomize -url jdbc:db2://localhost:50000/dbname -user USER_NAME -password PASSWORD
      SQLJTransactionTest_SJProfile0.ser
      ```

   b. Run the db2sqljcustomize tool to customize the SQLJ profiles. After you successfully run the db2sqljcustomize command, customized profiles exist in the directory from which you issued the command. If you run the db2sqljcustomize command from the directory that contains the serialized profiles that were not customized, the customized versions will overwrite previous versions that have the same file names.

      The recommended syntax for running the db2sqljcustomize command is:

      ```
      db2sqljcustomize -url JDBC_URL -user USER_NAME  -password PASSWORD
      [-rootpkgname PACKAGE_NAME] SERIALIZED_PROFILE1 SERIALIZED_PROFILE2 ...
      ```

      where:

      - *JDBC_URL* is the JDBC URL that is used to access the DB2 system where your tables reside.
      - *USER_NAME* is a valid user name for the DB2 system where your tables reside.
      - *PASSWORD* is the password for the specified user name.
      - *PACKAGE_NAME* is a valid partitioned data set (PDS) member name, up to seven characters long. Each of the four packages that are created by the profile customizer begin with this name and are appended with a number from 1 to 4. If you customize only one serialized profile, this value defaults to a shortened version of the serialized profile name and the -rootpkgname

parameter is not required. If you customize more than one serialized profile with the same command, there is no default value and the -rootpkgname parameter is required.

- *SERIALIZED_PROFILE#* is the name of the serialized profile that you are customizing.
  - To customize more than one serialized profile with the same command, list multiple files, separated by spaces.
  - Alternatively, you can specify the `-rootpkgname` parameter to customize more than one serialized profile with the same command.

Note: The following options provide more control over the customization process:

- `-automaticbind yes` specifies to run the DB2 SQLJ customizer against the serialized profiles to create static SQL in the database that the database will use at run time. The customization phase creates four database packages that contain static SQL, one for each isolation level.
- `-onlinecheck NO and -bindoptions "VALIDATE RUN"` specifies settings to bypass errors during a profile customization and ensure a successful customization.

7. Update the JAR file for the enterprise beans with the serialized profiles.
8. Use the `jar` command to replace the serialized profiles in your JAR file with the customized profiles.

   Note: The customized files must be placed in a location that is part of the application class path, and they must exist ahead of the serialized profiles that are not customized in your JAR file. If you decide to replace the serialized profiles in your JAR file, maintain the directory structure in which the profiles exist.
9. Package the JAR file for the enterprise bean, servlets, and serialized profiles into an enterprise archive (EAR) file.
10. Install the application in the application server.

    Note: Do not select **Deploy enterprise beans** during the application installation process in the administrative console. If you redeploy the enterprise beans from the administrative console, you will lose the customization changes that you have made.

SQLJ profiles and pureQuery bind files settings

Use this page to do customization and binding for the Structured Query Language in Java (SQLJ) profiles for DB2 that are included in this application. You can also use this page to do binding for pureQuery bind files in the application. You can view SQLJ profiles for other database types, but you cannot change these profiles. PureQuery bind files are only valid for DB2. Use SQLJ or pureQuery to develop data access applications that connect to DB2 databases. SQLJ is a set of programming extensions that enable a programmer to use the Java programming language to embed statements that provide SQL database requests. PureQuery provides an alternate set of APIs that can be used instead of JDBC to access the DB2 database.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **SQLJ profiles and pureQuery bind files**.

Advantages of developing applications with SQLJ include improved performance and a shorter, more efficient development cycle. With SQLJ, you can:

- Improve performance by using static SQL statements.
- Reduce the development cycle by:
  - Writing less code with the simpler SQLJ syntax, which reduces the amount of code that is required to execute statements, and set and retrieve parameters.
  - Detecting programming errors earlier in the development phase with the online check function, which performs data type and schema validation. Activate this function by running it as an option with the **db2sqljcustomize** command. See the DB2 documentation for a complete description of the SQLJ customize command.

DB2 pureQuery run time is an alternative set of APIs to JDBC or SQLJ. Advantages of developing applications with pureQuery include allowing SQL execution to be either dynamic or static. In addition to improved performance by using static SQL statements, pureQuery has better problem determination and diagnosis because it allows for errors at the DB2 server to be related back to application artifacts rather than to SQL that was generated by an application generator.

Customize and bind profiles:

Specifies that the application server processes the SQLJ profiles that you select from this application.

Note: This selection does not apply to pureQuery. If selected, this option is ignored when processing pureQuery bind files.

By default, one DB2 package is created in the database for each isolation level. The customization process augments the profile or profiles with information that is specific for the DB2 database for use at run time. Typically, the customization process should run after the SQLJ application has been translated and before the application is started. If you do not run the customization step, the SQLJ application uses dynamic SQL like a JDBC application.

Binding DB2 SQLJ profiles involves the process of binding the customized SQLJ profiles to the DB2 database.

Bind packages:

Specifies that the application server binds the SQLJ profiles that you select to the DB2 database server.

Note: This selection does not apply to pureQuery. If selected, this option is ignored when processing pureQuery bind files.

Bind packages from the SQLJ application that have already been customized.

Select and order the profiles to customize/bind:

Specifies the profiles to process from the list that is provided.

- Select a profile or group of profiles from the **Available profiles**, and click **Add** to add the profile that is selected to **Selected Profiles**.
- Select a profile or group of profiles from the **Selected Profiles**, and click **Remove** to add the profile that is selected to **Available profiles**.

When SQLJ or pureQuery profiles have been added to **Selected Profiles**, select profiles from that list and use **Move Up** or **Move Down** to change the order in which the profiles are processed.

Customize/bind the selected SQLJ profiles as a group:

Specifies that the application server creates a .grp file that contains the SQLJ profiles that you selected.

Note: This selection does not apply to pureQuery. If selected, this option is ignored when processing pureQuery bind files.

When you click **OK**, there is an option on the next page to download the `.grp` file.

Use a profile group file to specify profiles to customize/bind:

Specifies a profile group file from the local file system to customize or bind.

Database URL:

Specifies the URL of the database to which the profile or profiles are bound.

The typical syntax is:
`jdbc:db2://host_name:port_name/database_name`

User:

Specifies the user ID for the database administrator on the server where the database is located.

Password:

Specifies the password for the database administrator on the server where the database is located.

Additional options:

Specifies additional options to use during the customization and bind processes.

Options for pureQuery binding uses the following syntax:
`-bindoptions "BLOCKING NO"`

For more information about pureQuery bind options, refer to the DB2 pureQuery Bind Utility topic.

Class path:

Specifies the class path where the `sqlj.zip`, and `db2jcc.jar` or `db2jcc4.jar` files for SQLJ are located. Specifies the class path where the `pdq.jar`, `pdqmgt.jar`, `db2jcc.jar`, and `db2jcc_license_cisuz.jar` files for pureQuery are located.

Download SQLJ profile group

Use this panel to download the group file for the Structured Query Language in Java (SQLJ) profiles that are bound together as a single package on the DB2 database server. You can use the file when performing future customization or binding work on the application. Click the link that is provided to download the profile group to your local file system. The group file has a filename extension of .grp and a HTTP Content-Type of text/plain.. Your web browser settings might cause the browser to display the file contents rather than prompting you for a download destination. If this happens, you can manually copy and paste the contents into your own .grp file.

Note: This topic does not apply to IBM Optim PureQuery Runtime. IBM Optim PureQuery Runtime does not support binding pureQuery bind files as a group.

Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***app_name*** > **SQLJ profiles and pureQuery bind files**. When you are selecting the profiles to customize and bind, select **Customize/bind the selected SQLJ profiles as a group** to view this console panel.

Review results

Use this panel to review the results from the customization and binding process for the Structured Query Language in Java (SQLJ) profiles or pureQuery bind files. Use SQLJ or IBM Optim PureQuery Runtime to develop data access applications that connect to DB2 databases. SQLJ is a set of programming extensions that enable a programmer to use the Java programming language to embed statements that provide SQL (Structured Query Language) database requests. IBM Optim PureQuery Runtime provides an alternate set of APIs that can be used instead of JDBC to access the DB2 database.

Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **SQLj profiles and pureQuery bind**. Select profiles to customize and bind, complete the necessary fields, and click **OK** to view this console panel.

Review results:

Displays the results of the customization and bind process. The field shows information that is received from the database and summary statements from the application server.

Using embedded SQLJ with the DB2 for z/OS Legacy driver

Structured Query Language in Java (SQLJ) is a set of programming extensions that enable a programmer, using the Java programming language, to embed statements that provide Structured Query Language (SQL) database requests. You can use the DB2 for z/OS Legacy driver with your data access applications.

About this task

Notes:

1. To use SQLJ with WebSphere Application Server for z/OS and the DB2 for z/OS Legacy Driver, install DB2 APAR PQ76442.
2. Container Managed Persistence (CMP) beans generated using SQLJ are not supported by the DB2 for z/OS Legacy Driver. Use the DB2 Universal Driver for CMPs that are generated using SQLJ.

Following are the steps required to develop applications with SQLJ that run on WebSphere Application Server for z/OS v6.0 using the DB2 for z/OS Legacy driver.

Procedure

1. Design your application in Rational Application Developer according to your requirements, using SQLJ when necessary. For example, if you develop a bean called Test that uses BMP, code TestBean.sqlj (instead of TestBean.java).
   a. From your DB2 for z/OS installation, copy the `db2sqljclasses.zip` file to a directory on your workstation, then modify the Java Build Path of your EJB Java archive (JAR) project to include the `db2sqljclasses.zip` file.
   b. Translate your SQLJ code according to the following steps:
      1) Locate your SQLJ file, then use ASCII mode transfer to FTP it to an HFS in your z/OS environment.
      2) Use the sqlj command to translate your SQLJ code into Java code. Two files are produced, one with a `.java` extension and the other with an `.ser` extension.

         ```
         sqlj -compile=false SQLJ_FILE_NAME
         ```

      3) Use ASCII mode transfer for the `.java` file and BINARY mode transfer for the `.ser` file to move these files back to the directory on your workstation where the SQLJ file resides.
      4) Refresh the project.
   c. Generate deployment code for your application.
   d. Export your EAR file.
2. Install your application
   a. Create a data source with the DB2 for zOS Local JDBC Provider (RRS). When you define your JDBC Provider and data source, the default values are sufficient for providing SQLJ support.
   b. Install your application into WebSphere Application Server.

      Use the data source you created in Step 1 to resolve your resource references.
3. Customize your serialized profiles When you generate your deployment code, serialized profiles, or files with an `.ser` extension, that are specific to your application, are created. These profiles must be customized in a z/OS environment before they can be used.
   a. Use binary transfer to transfer the serialized profiles to the z/OS environment on which you installed your application. Alternatively, use the Java jar command to extract the serialized profiles from the EJB JAR file in your installed EAR directory.

b. Use the db2profc command to customize your serialized profiles. You can get information about the various options associated with this command from the DB2 documentation; however, here are the minimum requirements to customize your profile:

   ```
   db2profc -pgmname=PROGRAM_NAME PROFILE_NAME
   ```

   - Where:
     - *PROGRAM_NAME* must be a valid MVS PDS member name, and can be up to seven characters.
     - *PROFILE_NAME* is the name of the serialized profile that you want to customize. You must run db2profc one time for each profile.
   - The profile customizer creates four DBRM data sets in the PDS *USERNAME*.DBRMLIB.DATA. The member names of the DBRMs begin with what you specified as *PROGRAM_NAME*.
   - Ensure that your CLASSPATH environment variable includes:
     - The location of the serialized profile
     - The EJB JAR file in your installed EAR directory
   - Allocate a PDS to contain the DBRMs that are created. Name this PDS *USERNAME*.DBRMLIB.DATA, where *USERNAME* is the user who implements the db2profc command.

     The following fields are an example:

     ```
     Space units=TRACK
     Primary quantity=15
     Secondary quantity=5
     Directory blocks=10
     Record format=FB
     Record length=80
     Block size=27920
     Data set name type=PDS
     ```

c. Place the existing serialized profiles, which are now customized, into a location that is part of the application classpath and that is ahead of the serialized profiles that exist in your EJB JAR file.

   The output of the DB2 profile customizer and the input file have the same name. Move the output file ahead of the original serialized profile in the classpath. Alternatively, you can move the customized profile into the EJB JAR file, replacing the original. It is recommended that you replace the original file.

   IMPORTANT: If you run the db2profc command from the directory where the serialized profile exists, the profile customizer overwrites the serialized profile. Because you need only the customized version after the profile customizer has run, this is not a problem.

d. Bind your DBRMs into a package.

   Note: You must create your database tables before binding your DBRMs. If you do not, the bind job fails.

   The db2profc customization command creates a series of DBRMs that must be bound into packages. For each customized profile, four DBRMs are created.

   These DBRMs:

   - Are located in *USERNAME*.DBRMLIB.DATA
   - All have names that begin with what you specified as *PROGRAM_NAME*
   - Are numbered from 1-through-4

     For example, if you log in as IBMUSER, and you specify -pgmname=TESTBMP, then run the db2profc command, the four data sets, TESTBMP1, TESTBMP2, TESTBMP3, AND TESTBMP4 are created and placed in the PDS IBMUSER.DBRMLIB.DATA.

     These data sets must be bound into packages with isolation of UR, CS, RS, and RR. You must run a bind for each serialized profile that you customize.

e. After you bind all of the DBRMs into packages, bind the packages into a plan. Name the plan whatever you like.

IMPORTANT: You must also include the JDBC packages in the package list (PKLIST) of your new plan. The default names for the JDBC packages to include are DSNJDBC.DSNJDBC1, ..., DSNJDBC.DSNJDBC4. If your installation did not use the default names for the JDBC packages, contact your DB2 administrator to determine the names of the JDBC packages that you need to include.

Following is a sample job used to bind a new plan.

- One serialized profile was created while logged on as IBMUSER.
- **-pgmname=TESTBMP** was specified to run db2profc.
- The new plan is named SQLJPLAN.

```
//BBOOLS  JOB (516B,1025),'IBMUSER',MSGCLASS=H,CLASS=A,PRTY=14,
//         NOTIFY=&SYSUID,TIME=1440,USER=IBMUSER,PASSWORD=IBMUSER,
//         MSGLEVEL=(1,1)
//*******************************************************************
//BINDOLS  EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB  DD  DSN=IBMUSER.DBRMLIB.DATA,DISP=SHR
//*        DD  DSN=MVSDSOM.DB2710.SDSNDBRM,DISP=SHR
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSTSIN  DD  *

 DSN SYSTEM(DB2)

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP1) -
     VALIDATE(BIND) -
     ISOLATION(UR) -
     SQLERROR(NOPACKAGE) -

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP2) -
     VALIDATE(BIND) -
     ISOLATION(CS) -
     SQLERROR(NOPACKAGE) -

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP3) -
     VALIDATE(BIND) -
     ISOLATION(RS) -
     SQLERROR(NOPACKAGE) -

 BIND -
     PACKAGE(TESTBMP) -
     QUALIFIER(IBMUSER) -
     MEMBER(TESTBMP4) -
     VALIDATE(BIND) -
     ISOLATION(RR) -
     SQLERROR(NOPACKAGE) -


 BIND PLAN(SQLJPLAN)                -
  QUALIFIER(IBMUSER)                -
  PKLIST(TESTBMP.*                  -
         DSNJDBC.*                ) -
  ACTION(REPLACE) RETAIN            -
```

```
  VALIDATE(BIND)

END
/*
```

f. Grant the appropriate authority to your new plan. Use an interface to DB2, such as SPUFI, to grant the authority. Issue this command:

```
GRANT EXECUTE ON PLAN PLANNAME TO APPSERVERID
```

Where:

- *PLANNAME* is the name of the plan that you bound.
- *APPSERVERID* is the ID under which WebSphere Application Server runs; for example, CBSYMSR1.

4. Configure your data source to use your new plan
   a. From the WebSphere Application Server for z/OS Administrative Console, navigate to your Data Source and select Custom Properties.
   b. Select the Custom Property **planName**.
   c. Update the value of **planName** with what you named your plan when it was bound.
   d. Set **enableSQLJ** to **true**.
5. Stop and restart your server.
6. Run your application.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

  In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root

Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is `/usr/lpp/zWebSphere/V8R5`.

Installing a resource adapter archive

The application server uses the classes and other code that comprise a resource adapter archive (RAR) to support the resource adapters that you configure.

Before you begin

A RAR file, which is often called a Java EE Connector Architecture (JCA) connector, must comply with the JCA Specification. You can meet these requirements by using a supported assembly tool to assemble a collection of Java archive (JAR) files, other runnable components, and utility classes into a deployable resource adapter archive (RAR). You can then install the RAR file in the application server.

When you are using optimized local adapters, set up your server environment in the daemon group before installing the `ola.rar` file in the application server. The `ola.rar` file is located in the `WAS_HOME/installableApps` directory.

You can also use the wsadmin scripting file, olaRar.py, to install the RAR file.

To read about setting up your server environment, see the topic, Enabling the server environment to use optimized local adapters. You can read about the `olaRar.py` script in the topic, olaRar.py scripting file.

About this task

A resource adapter archive provides the classes and other code to support a resource adapter for access to a specific EIS, such as the Customer Information Control System (CICS). Therefore, you can only configure resource adapters for an EIS after you install the appropriate RAR file.

Important: When you use the **Install RAR** dialog to install a RAR file, the scope you define on the Resource Adapters page has no effect on where the RAR file is installed. You can install RAR files only at the node level, which you specify on the Install RAR page. To set the scope of an RAR file to a specific cluster, or server, after you install the RAR file at each node level, create a copy of the RAR file with the appropriate cluster or server scope.

Procedure

1. Navigate to the **Resource adapter** panel. Click **Resources** > **Resource Adapters** > **Resource adapters**.
2. Install a new resource adapter archive.
   a. Click **Install RAR**. A dialog opens for installing a RAR file and configuring the associated resource adapter. Only click **New** if you want to configure a new resource adapter for a previously installed RAR file.
   b. Browse to find the appropriate RAR file.
      - If your RAR file is located on your local workstation, select **Local path**, and browse to find the file.
      - If your RAR file is located on your server, select **Remote file system**, and specify the fully qualified path to the file.
   c. Click **Next**.

3. Configure the resource adapter name and any other properties needed under *General Properties*. For more details on the settings that you can configure, such as the J2C connection factories, see the topics Installing resource adapters within applications and Configuring resource adapters.
4. Click **OK**.
5. Optional: Create a copy of the RAR file with a different scope level. After you install the RAR file at each node level, you can create another copy of the file that has a specific server or cluster as the scope for that file.

   Note:

   - If you do not create a copy of your RAR at the cluster scope, then you must create identical factories (connection factories, admin object, and activation specifications) at the node level for each of your nodes in the cluster. By creating the copy of your RAR, you provide a placeholder for your factories and circumvent the need to create identical factories at the node level for each of your nodes in the cluster.
   - You must still install the RAR binaries (files, such as jars and xml deployment files) on each node for the RAR to operate successfully.

   a. Click **Resources**.
   b. Click **Resource Adapters**.
   c. Select the scope level and then click **NEW**.
   d. Choose the RAR file from the installed archive path.
   e. Click **OK**.

Results

You have installed a resource adapter archive that provide access to the EIS when it is properly configured. If you must configure more settings, or change some settings that were configured during the installation process, refer to the topic on configuring a resource adapter in the administrative console for more information.

Installing resource adapters embedded within applications

Install resource adapters in your applications so they can access outside data sources.

Before you begin

The JCA Version 1.6 specification adds support for Java annotations in RAR modules. For more information on annotation support see the topic, JCA 1.6 support for annotations in RAR modules.

About this task

Procedure

1. Assemble an application with RAR modules in it. See the topic Assembling applications for more information.
2. Install the application. Follow the steps in the topic Installing a new application.

   In the **Map modules to servers** step, specify target servers or clusters for each RAR file. Be sure to map all other modules that use the resource adapters defined in the RAR modules to the same targets. Also, specify the web servers as targets that serve as routers for requests to this application. The plug-in configuration file (`plugin-cfg.xml`) for each web server is generated based on the applications that are routed through it.

   In the **Metadata for modules** step of installing an application, you can set or unset the metadata-complete flag as discussed in the topic, JCA 1.6 support for annotations in RAR modules.

Note: When installing a RAR file on a server, the application server looks for the manifest (MANIFEST.MF) for the connector module. The application server first looks for the RAR file's `connectorModule.jar` file and loads the manifest from the `connectorModule.jar` file. If the class path entry is in the manifest from the `connectorModule.jar`file, the RAR uses that class path.

To ensure that the installed connector module finds the classes and resources that it needs, check the Class path setting for the RAR using the administrative console. For more information on how to check this setting, see the topics Resource adapter settings and WebSphere relational resource adapter settings.

3. Click **Finish** > **Save** to save the changes.
4. Create connection factories for the newly installed application.

   See the topic, Configuring connection factories for resource adapters within applications to view the steps to complete this step.

Results

Note: A given native library can only be loaded one time for each instance of the Java virtual machine (JVM). Because each application has its own class loader, separate applications with embedded RAR files cannot both use the same native library. The second application receives an exception when it tries to load the library.

If any application deployed on the application server uses an embedded RAR file that includes native path elements, then you must always ensure that you shut down the application server cleanly, with no outstanding transactions. If the application server does not shut down cleanly it performs *recovery* upon server restart and loads any required RAR files and native libraries. On completion of recovery, do not attempt any application-related work. Shut down the server and restart it. No further recovery is attempted by the application server on this restart, and normal application processing can proceed.

Install RAR

Use this page to install a resource archive (RAR) file in one of two ways. You can either upload a RAR file from the local file system, or specify an existing RAR file on a server. The RAR file must be installed at the node level, and you can select the node on this page.

To view this page in the administrative console click **Resources** > **Resource Adapters** > **Resource Adapters** > **Install RAR**.

For information about installing a resource adapter, see the topic, Installing a resource adapter archive (RAR) file.

Scope

Specifies the scope of the resource adapter. Only applications that are installed within this scope can use this adapter.

Local file system

Specifies the path of a RAR that resides on the same server as the console.

Information	Value
Data type	String

Remote file system

Specifies the path of a RAR that resides on one of the nodes of the cell.

Information	Value
Data type	String

Chapter 42. Deploying EJB applications

This page provides a starting point for finding information about enterprise beans.

Based on the Enterprise JavaBeans (EJB) specification, enterprise beans are Java components that typically implement the business logic of Java Platform, Enterprise Edition (Java EE) applications as well as access data.

Deploying EJB 3.x enterprise beans

EJB module settings

Use this page to configure and manage a specific deployed EJB module.

Note: You cannot start or stop an individual EJB module for modification. You must start or stop the appropriate application entirely.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > **application_name** > **Manage Modules** > **module_name**.

Attention: If an application is running, changing an application setting causes the application to restart. On stand-alone servers, the application restarts after you save the change. On multiple-server products, the application restarts after you save the change and files synchronize on the node where the application is installed. To control when synchronization occurs on multiple-server products, deselect **Synchronize changes with nodes** on the Console preferences page.

URI

Specifies location of the module relative to the root of the application EAR file. The URI must match the URI of a ModuleRef URI in the deployment descriptor of the deployed application (EAR).

Alternate deployment descriptor

Specifies an alternate deployment descriptor for the module as defined in the application deployment descriptor according to the Java Platform, Enterprise Edition (Java EE) specification.

Starting weight

Specifies the order in which modules are started when the server starts. The module with the lowest starting weight is started first.

If the application deployment descriptor specifies the `<initialize-in-order>true</initialize-in-order>` element, the default starting weights reflect the order that is specified in the deployment descriptor. Otherwise, the defaults are determined based on module type (RAR modules start before EJB modules, which start before web modules).

Information	Value
Data type	Integer
Default	5000
Range	Greater than 0

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
Refers to the installation root directory for Web Server Plug-ins.

profile_root
Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Deploying EJB modules

When you deploy an Enterprise JavaBeans (EJB) module, you install that module on a server that has been configured to support deployed modules.

Before you begin

Assemble one or more EJB modules, assemble one or more web modules, and assemble them into a Java EE application.

For an overview about the changes to the EJB deployment model for EJB 3.x, see the topic EJB 3.x deployment overview.

Procedure

1. Prepare the deployment environment. See the topic Preparing to host applications.
2. Update the configuration for each EJB module as needed for the deployment environment.
3. Required: If a module has dependencies on Java 5-specific extensions, such as generics, annotations, and so on, then you must run the EJBDeplolycommand-line tool separately and before installing the module or application containing it. This is because the administrative console and the wsadmin command-line tool do not allow for specifying the ejbdeploy -complianceLevel 5.0 option.

It is only necessary to run the EJBDeploy tool for EJB 2.1 modules containing entity beans.

4. Address potential interoperability issues.

   There can be unexpected results if a WebSphere stack product, or another product, that runs on a version of Application Server that does not support EJB 3.x attempts to remotely invoke a method on an EJB 3.x compliant enterprise bean on a separate server that is running a version Application Server that supports EJB 3.x. If these products attempt to invoke a method through the enterprise bean's EJB 3.x remote business interface, they might encounter exceptions that were introduced in EJB 3.x that will be pushed back to the environment that is not EJB 3.x compliant.

   This scenario could also be an issue for an administrator of an environment that includes a combination of stack products that contain a mixture of EJB 3.x compliant and non-compliant instances of Application Server.

   The following is a list of the exception classes that have been introduced in EJB 3.0:

   - javax.ejb.ConcurrentAccessException
   - javax.ejb.EJBAccessException
   - javax.ejb.EJBTransactionRequiredException
   - javax.ejb.EJBTransactionRolledbackException
   - javax.ejb.NoSuchEJBException

   a. Ensure that Application Server is updated to 7.0.0.3.

   b. Manually copy the <*app_server_root*>/runtimes/ejb3exceptions.jar file from Application Server to a directory on each of the stack products installations, or other product installations, that you will use as the EJB 3.x client.

   c. Ensure that the directory that contains the ejb3exceptions.jar file is in the class path. One possible location for the JAR file that would satisfy this requirement is the <*app_server_root*>/lib directory on a server that is not EJB 3.x compliant.

      Note: Just like the EJB thin client jars, if an update becomes available, users must copy the ejb3exceptions.jar file again after installing the version of the WebSphere Application Server containing the updated version.

5. Deploy the application. See the topic Deploying and administering enterprise applications.

What to do next

If you specify that the EJBDeploy tool be run during application installation and the installation fails with a NameNotFoundException message, ensure that the input Java archive (JAR) or enterprise archive (EAR) file does not contain source files. Either remove the source files or include all dependent classes and resource files on the class path. If there are source files in the input JAR or EAR file, the EJB deployment tools runs a rebuild before generating the deployment code.

If the module deploys successfully, test and debug the module. See the topic Diagnosing problems (using diagnosis tools).

EJB 3.0 and EJB 3.1 deployment overview

Learn about the Enterprise JavaBeans (EJB) 3.0 and 3.1 deployment model, including *Just-In-Time* (JIT) deployment.

All Java Enterprise Edition (Java EE) application server products have some form of EJB deployment phase in which your application is customized to run in that particular implementation of the application server. Typically, this is accomplished by a deployment tool that is specific to the application server and generates code to bridge your EJB interface and implementation code to the application server's implementation for an EJB container. Some application server products' deployment tools alter the bytecodes of your application classes, rather than generating code, but the end result is similar.

Application Server bridges your EJB interface with its implementation by generating code that encapsulates your EJB implementation classes, connecting them to Application Server's EJB container. This enables the EJB container to host your enterprise beans and provide services to them. If one or more of your enterprise beans has remote interfaces defined, Application Server generates additional code to provide the remote function.

For more information about packaging your EJB module, see the topic that covers the EJB 3.x module packaging overview.

EJBDeploy Tool

Historically, EJB deployment in the Application Server product has been performed by the EJBDeploy tool, which is included with WebSphere Application Server and packaged with the development tools for the WebSphere products.

The EJBDeploy tool introspects the external interfaces for your enterprise beans, generates the wrapper code as .java files, and compiles the code using the javac compiler to produce .class files that are packaged in your EJB module with your application code. The EJBDeploy tool also runs the rmic tool against the remote EJB interfaces in the application, producing additional *stub* and *tie* class files that interact with the Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP) Object Request Broker (ORB), providing remote object support.

For modules previous to EJB 3.0, you ran the EJBDeploy tool when you installed the application on Application Server or before you installed the application from the command-line tool or a development tool.

Just-In-Time (JIT) deployment

EJB 3.0 support in Application Server introduced a new feature called JIT deployment.

With JIT deployment, the EJB container dynamically generates the wrapper, stub, and tie classes in-memory when the application is running. Additionally, the web container and application client containers dynamically generate the stub class that is required for remote EJB invocations.

Effectively, this means that you do not need to process EJB 3.0 or 3.1 modules, web modules that invoke EJB 3.0 or 3.1 beans, or client modules that invoke EJB 3.0 or 3.1 beans through the EJBDeploy tool before you run them in Application Server.

createEJBStubs tool

In most cases the Just-In-Time deployment feature can dynamically generate the RMI-IIOP stub classes that are required for invocation of remote EJB interfaces. There are some instances in which these stub classes are not dynamically generated. For EJB 3.0 or 3.1 clients that are not running inside an EJB 3.x enabled web container, EJB container, or client container, you must generate the stub classes with the createEJBStubs tool and ensure that the generated stubs are available in the client environment's class path. Typically, you would accomplish this by copying the generated stubs to the location where the client's business interface class resides.

The createEJBStubs tool must be used to generate client-side stubs for the following environments:

- "Bare" Java Standard Edition (SE) clients, where a Java SE Java Virtual Machine (JVM) is the client environment.
- Containers in Application Server environments prior to Version 7 that do not have the Feature Pack for EJB 3.0 applied.
- Environments that are not WebSphere Application Server environments.

Interoperability

There can be unexpected results if a WebSphere stack product, or another product, that runs on a version of Application Server that does not support EJB 3.0 or 3.1 attempts to remotely invoke a method on an EJB 3.x compliant enterprise bean on a separate server that is running a version Application Server that supports EJB 3.0 or 3.1. If these products attempt to invoke a method through the enterprise bean's EJB 3.x remote business interface, they might encounter exceptions that were introduced in EJB 3.0 that will be pushed back to the environment that is not EJB 3.x compliant.

This scenario could also be an issue for an administrator of an environment that includes a combination of stack products that contain a mixture of EJB 3.x compliant and non-compliant instances of Application Server.

The following is a list of the exception classes introduced in EJB 3.0:

- javax.ejb.ConcurrentAccessException
- javax.ejb.EJBAccessException
- javax.ejb.EJBTransactionRequiredException
- javax.ejb.EJBTransactionRolledbackException
- javax.ejb.NoSuchEJBException

Refer to the EJB module deployment step to address potential interoperability issues.

EJB 2.x Modules

EJB 2.x modules that have been converted to be EJB 3.0 or EJB 3.1 modules should have all WebSphere Application Server generated files (including stub and tie classes) removed prior to EJB deployment in the Application Server product.

EJBDEPLOY relationships – troubleshooting tips

Use this information to troubleshoot information for EJBDEPLOY problems.

DB2 for z/OS Version 7.x

Problems might exist when EJBDeploy creates a data relationship in DB2 for z/OS Version 7.x. EJBDeploy creates a table with a composite of the two primary keys of the EJBs that are related to each other. If the composite keys are larger than 254 characters, DB2 for z/OS V7.x does not accept this relationship and the following error can occur:

```
DSNT408I SQLCODE = -613, ERROR:  THE PRIMARY KEY OR A UNIQUE CONSTRAINT
IS TOO LONG OR HAS TOO MANY COLUMNS
DSNT418I SQLSTATE   = 54008 SQLSTATE RETURN CODE
```

This problem can be seen when the primary keys that are created for the two related beans have primary keys that are strings. This results in the composite being made up of 2 varchar(250) primary keys for a total of 500, which is greater than 254 maximum in DB2 for z/OS version 7.x.

Things to consider when utilizing top-down mappings to ensure you do not experience this problem:

- Top-down mappings are a guideline and must be reviewed with the DBA.
- Schemas that are created top-down by EJBDeploy are designed only for testing, and as a guideline for the actual schema required. The use of the meet-in-the-middle mapping does not present this problem.
- The composite key constraint problem is not experienced when using DB2 V8, which has 2K maximum key lengths.

EJBDEPLOY_JVM_ARGS:

Set the EJBDEPLOY_JVM_ARGS property to override Java virtual machine (JVM) options that are passed to the code that deploys EJBs (ejbdeploy.sh). Set this property in one of the following locations: *deploymentmanager*/bin/setupCmdLine.sh or *appServerHome*/bin/setupCmdLine.sh

For example, the following specifies that unqualifed SQL should be generated:

```
export EJBDEPLOY_JVM_ARGS="-DEJBDEPLOY_GENERATE_UNQUALIFIED_SQL=true"
```

The converter that is defined for the primary key is not invoked on its foreign key value

The mapping for primary key fields to database columns may use a converter to transform the key values. If a container-managed persistence (CMP) bean uses a converter to map its primary key, and that bean has a relationship where the bean at the other end holds a foreign key, the mapping for the foreign key will not use the converter.

The following errors might occur, indicating that the converter defined for the primary key is not invoked on its foreign key value. During the run of the ejbDeploy command , you receive the following message:

```
No type mapping defined for Java datatype1 to Database datatype2
```

During run time, the application does not find the CMP bean at the other end of the relationship.

To work around this limitation, define your own foreign key in the database table, and create a mapping that uses the same converter as defined for the primary key on the enterprise beans at the other end of its relationship.

Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories imply specific default directory locations. This article describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
: Refers to the top directory for a WebSphere Application Server node.

  The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are *was.install.root* and *WAS_HOME*.

  The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
: Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

  The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

  The default is /wasv8config/*cell_name*/*node_name*.

plug-ins_root
: Refers to the installation root directory for Web Server Plug-ins.

profile_root
: Refers to the home directory for a particular instantiated WebSphere Application Server profile.

  Corresponding product variables are *server.root* and *user.install.root.*

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root

Refers to the root directory for product code installed with SMP/E or IBM Installation Manager.

The corresponding product variable is *smpe.install.root*.

The default is /usr/lpp/zWebSphere/V8R5.

Chapter 43. Deploying messaging resources

This page provides a starting point for finding information about deploying asynchronous messaging resources for enterprise applications with WebSphere Application Server.

WebSphere Application Server supports asynchronous messaging based on the Java Message Service (JMS) and the Java EE Connector Architecture (JCA) specifications, which provide a common way for Java programs (clients and Java EE applications) to create, send, receive, and read asynchronous requests, as messages.

JMS support enables applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). Some messaging providers also allow WebSphere Application Server applications to use JMS support to exchange messages asynchronously with non-JMS applications; for example, WebSphere Application Server applications often need to exchange messages with WebSphere MQ applications. Applications can explicitly poll for messages from JMS destinations, or they can use message-driven beans to automatically retrieve messages from JMS destinations without explicitly polling for messages.

WebSphere Application Server supports the following messaging providers:

- The WebSphere Application Server default messaging provider (which uses service integration as the provider).
- The WebSphere MQ messaging provider (which uses your WebSphere MQ system as the provider).
- Third-party messaging providers that implement either a JCA Version 1.5 resource adapter or the ASF component of the JMS Version 1.0.2 specification.

Deploying enterprise applications

You can deploy an enterprise application to use JMS, or to use message-driven beans.

About this task

Enterprise applications can use JMS APIs directly to explicitly poll for messages on a JMS destination, then retrieve messages for processing by business logic beans (enterprise beans).

Message-driven beans can also be used as asynchronous message consumers. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination.

Procedure

- Deploy an enterprise application to use JMS.
- Deploy an enterprise application to use message-driven beans.

Deploying an enterprise application to use JMS

You can deploy an enterprise application to use JMS.

About this task

This task description assumes that you have an .EAR file, which contains an application enterprise bean with code for JMS, that can be deployed in WebSphere Application Server.

To deploy an enterprise application to use JMS, complete the following steps:

Procedure

1. Configure the deployment attributes for the application, as described in Assembling applications.
2. Use the WebSphere Application Server administrative console to install the application.

   This stage is a standard WebSphere Application Server task, as described in Installing applications.

Deploying enterprise applications developed as message-driven beans

You can deploy an enterprise application developed as a message-driven bean in WebSphere Application Server.

About this task

Message-driven beans can be used as asynchronous message consumers. When a message arrives at the destination, the EJB container invokes the message-driven bean automatically without an application having to explicitly poll the destination.

Procedure

- Deploy an enterprise application to use message-driven beans with JCA 1.5-compliant resources. You can configure message-driven beans as listeners on a Java EE Connector Architecture (JCA) 1.5 resource adapter, such as the default messaging provider or the WebSphere MQ messaging provider in WebSphere Application Server.
- Deploy an enterprise application to use message-driven beans with listener ports. Listener ports are stabilized. For more information, read the article on stabilized features. You should only deploy your application against a listener port for compatability with existing message-driven bean applications. Otherwise, you should deploy your application against JCA 1.5-compliant resources.

Deploying an enterprise application to use message-driven beans with JCA 1.5-compliant resources

Message-driven beans can be configured as listeners on a Java EE Connector Architecture (JCA) 1.5 resource adapter, such as the default messaging provider or the WebSphere MQ messaging provider in WebSphere Application Server.

Before you begin

This task assumes that you have an EAR file that contains an enterprise application, developed as a message-driven bean, that can be deployed in WebSphere Application Server.

Note: You can continue to deploy message-driven beans against a listener port. You might want to do this for compatability with existing message-driven bean applications. However, listener ports are stabilized, and you should plan to migrate all your message-driven beans to use JCA 1.5-compliant or 1.6-compliant resources.

About this task

You deploy message-driven beans against JCA 1.5-compliant resources, and configure the resources as deployment descriptor attributes and (for EJB 3) as annotations.

Procedure

1. For each message-driven bean in the application, create a J2C activation specification.
2. For each message-driven bean in the application, configure the J2C deployment attributes, as described in “Configuring deployment attributes for a message-driven bean against JCA 1.5-compliant resources” on page 2085.
3. Use the WebSphere Application Server administrative console to install the application.

Configuring deployment attributes for a message-driven bean against JCA 1.5-compliant resources:

You can configure the message-driven bean deployment attributes for a Java EE Connector Architecture (JCA) 1.5-compliant enterprise application, to override the deployment attributes defined within the application EAR file.

Before you begin

This task assumes that you have an EAR file that contains an enterprise application, developed as a message-driven bean, that can be deployed in WebSphere Application Server.

Note: You can continue to configure message-driven beans against a listener port. You might want to do this for compatability with existing message-driven bean applications. However, listener ports are stabilized, and you should plan to migrate all your message-driven beans to use JCA 1.5-compliant or 1.6-compliant resources.

About this task

You configure the deployment attributes of a message-driven bean application by using an assembly tool. Detailed steps given in this task are for Rational Application Developer, but other tools have very similar steps.

Procedure

1. Start your assembly tool.
2. Edit the application EAR file. For example, use the Rational Application Developer import wizard to import the EAR file into the assembly tool. To start the import wizard:
   a. Click **File** > **Import** > **EAR file**.
   b. Click **Next**, then select the EAR file.
   c. Click **Finish**.
3. Open the deployment attributes for editing. In the Java EE Hierarchy view, right-click the EJB module for the message-driven bean then click **Open With** > **Deployment Descriptor Editor**. A property dialog notebook for the message-driven bean is displayed in the property pane.
4. Review and, if needed, change the deployment attributes.
   a. In the property pane, select the **Bean** tab.
   b. Under **Activation Configuration**, review the attributes.

      Note: For EJB 3 message-driven beans, you can instead use an EJB 3 annotation to configure the activation configuration properties. Do not use an EJB 3 annotation to change or replace what is specified in the bean deployment descriptor. If an activation configuration property is specified in both places, the value used is the one that is given in the deployment descriptor.

      `acknowledgeMode`
      This attribute determines how the session acknowledges any messages it receives.
      `Auto Acknowledge`
      The session automatically acknowledges delivery of each message.
      `Dups OK Acknowledge`
      The session lazily acknowledges the delivery of messages. This setting is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumer applications that are tolerant of duplicate messages.

      `destinationType`
      This attribute determines whether the message-driven bean uses a queue or topic destination.
      `Queue` The message-driven bean uses a queue destination.

Topic The message-driven bean uses a topic destination.

subscriptionDurability
This attribute determines whether a JMS topic subscription is durable or nondurable.

Durable
A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription messages until they are received by the subscription or until they expire.

Nondurable
Nondurable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A nondurable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created.

messageSelector
This attribute determines the JMS message selector that is used to select which messages the message-driven bean receives. For example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

c. Specify bindings deployment attributes.

Under **WebSphere Bindings**, select the **JCA Adapter** option then specify the bindings deployment attributes:

ActivationSpec JNDI name
This attribute specifies the JNDI name of the activation specification that is used to deploy this message-driven bean. This name must match the name of an activation specification that you define to WebSphere Application Server.

ActivationSpec Authorization Alias
This attribute specifies the name of an authentication alias used for authentication of connections to the JCA resource adapter. An authentication alias specifies the user ID and password that is used to authenticate the creation of a new connection to the JCA resource adapter.

Destination JNDI name
This attribute specifies the JNDI name that the message-driven bean uses to look up the JMS destination in the JNDI namespace.

5. Save your changes to the deployment descriptor:
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.
6. Verify the archive files.
7. From the pop-up menu for the project, click **Deploy** to generate EJB deployment code.
8. Optional: Test your completed module on a WebSphere Application Server installation.

   Right-click a module, click **Run on Server**, then follow the instructions in the displayed wizard.

   Restriction: **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only. You cannot deploy remotely to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

Important: Use **Run on Server** for unit testing only. When an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that will run the application; for example, use the administrative console as described in Deploying and managing applications.

Configuring servant regions for message-driven beans with JCA version 1.5 resource adapters:

For any message-driven beans (MDBs) that are not driven by asynchronous messaging or the IMS Connect for Java (IC4J) resource adapter, you must redefine the servant region for those beans that are connected to Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) Version 1.5 resource adapters that support inbound message processing.

About this task

If the application server is configured with one or more message-driven beans (MDBs) that do not use asynchronous messaging and are bound to a Java EE Connector Architecture (JCA) Version 1.5 resource adapter, and the resource adapter supports inbound message processing, the adapter must be defined to run only a maximum of 1 servant region. If you use the IMS Connect for Java (IC4J) resource adapter or an adapter that is driven by asynchronous messaging, you can configure the servant regions through the adapter configuration.

Note: If the application server is configured with one or more message-driven beans (MDBs) that are bound to service integration messaging or WebSphere MQ messaging using a resource adapter, which means that the MDBs use activation specifications rather than listener ports, then you must enable the Control Region Adjunct (CRA) process for the application server. To include starting the CRA process as part of starting an application server, either select **Enable JCA based inbound message delivery** on the JMS provider settings panel, or use the manageWMQ command.

Procedure

1. Open the administrative console.
2. Select **Servers > Server Types > WebSphere application servers**.
3. Select the server you want to configure.
4. Under Server Infrastructure, expand **Java and Process Management**.
5. Select **Process Definition**.
6. Select **Servant**.
7. Under Additional Properties, select **Environment Entries**. This causes a panel with the heading Application servers > *server* > Process Definition > Servant > Custom Properties to appear.
8. Select **New**. A panel with three General Properties fields appears. This is where you create the environment variable for controlling the number of servant regions.
9. In the Name field, enter **wlm_maximumSRcount**.
10. In the Value field, enter **1**. A servant can be configured to a maximum of 1 servant region by using the administrative console.
11. Select **Apply**.

Deploying an enterprise application to use message-driven beans with listener ports

Define the listener ports for your application. For each message-driven bean in the application, configure the deployment attributes to match the listener port definitions. Install the application, specifying the name of the listener port to use for late responses.

Before you begin

Listener ports are stabilized. For more information, read the article on stabilized features. You should only deploy your application against a listener port for compatability with existing message-driven bean applications. Otherwise, you should deploy your application against JCA 1.5-compliant resources.

If you have existing message-driven beans that use the WebSphere MQ messaging provider (or a compliant third-party JMS provider) with listener ports, and instead you want to use EJB 3 message-driven beans with listener ports, these new beans can continue to use the same messaging provider.

This task assumes that you have an EAR file, which contains an enterprise application developed as a message-driven bean, that can be deployed in WebSphere Application Server.

Procedure

1. Define the listener ports for your application, by using the WebSphere Application Server administrative console as described in Creating a new listener port.
2. For each message-driven bean in the application, configure the deployment attributes to match the listener port definitions, as described in "Configuring deployment attributes for a message-driven bean against a listener port."
3. Install the application, as described in Installing enterprise application files with the console.

   When you install the application, you are prompted to specify the name of the listener port that the application is to use for late responses. Select the listener port, then click **OK**.

Configuring deployment attributes for a message-driven bean against a listener port:

You can configure the message-driven beans deployment attributes for an enterprise bean, to override the deployment attributes defined within the application EAR file.

Before you begin

Listener ports are stabilized. For more information, read the article on stabilized features. You should only configure your application against a listener port for compatability with existing message-driven bean applications. Otherwise, you should configure your application against JCA 1.5-compliant resources.

If you have existing message-driven beans that use the WebSphere MQ messaging provider (or a compliant third-party JMS provider) with listener ports, and instead you want to use EJB 3 message-driven beans with listener ports, these new beans can continue to use the same messaging provider.

This task assumes that you have an EAR file that contains an enterprise application, developed as a message-driven bean, that can be deployed in WebSphere Application Server.

About this task

You configure the deployment attributes of a message-driven bean application by using an assembly tool. Detailed steps given in this task are for Rational Application Developer, but other tools have very similar steps.

Procedure

1. Start your assembly tool.

2. Edit the application EAR file. For example, use the Rational Application Developer import wizard to import the EAR file into the assembly tool. To start the import wizard:
   a. Click **File** > **Import** > **EAR file**.
   b. Click **Next**, then select the EAR file.
   c. Click **Finish**.
3. Open the deployment attributes for editing. In the Java EE Hierarchy view, right-click the EJB module for the message-driven bean then click **Open With** > **Deployment Descriptor Editor**. A property dialog notebook for the message-driven bean is displayed in the property pane.
4. Specify general deployment attributes.
   a. In the property pane, select the **Bean** tab.
   b. On the main panel, configure the **Transaction type** attribute.

      This attribute determines whether the message-driven bean manages its own transactions, or whether the container manages transactions on behalf of the bean.

      Bean The message-driven bean manages its own transactions.

      Container
      The container manages transactions on behalf of the bean.
5. Under **Activation Configuration**, review the following attributes:

   Note: For EJB 3 message-driven beans, you can instead use an EJB 3 annotation to configure the activation configuration properties. Do not use an EJB 3 annotation to change or replace what is specified in the bean deployment descriptor. If an activation configuration property is specified in both places, the value used is the one that is given in the deployment descriptor.

   acknowledgeMode
   This attribute determines how the session acknowledges any messages it receives.

   Auto Acknowledge
   The session automatically acknowledges delivery of each message.

   Dups OK Acknowledge
   The session lazily acknowledges the delivery of messages. This setting is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumer applications that are tolerant of duplicate messages.

   destinationType
   This attribute determines whether the message-driven bean uses a queue or topic destination.

   Queue The message-driven bean uses a queue destination.

   Topic The message-driven bean uses a topic destination.

   subscriptionDurability
   This attribute determines whether a JMS topic subscription is durable or nondurable.

   Durable
   A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription messages until they are received by the subscription or until they expire.

   Nondurable
   Nondurable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

   A nondurable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created.

messageSelector
This attribute determines the JMS message selector that is used to select which messages the message-driven bean receives. For example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

6. Specify the bindings deployment attribute.
   a. Under **WebSphere Bindings**, specify the following attribute:
      Listener port name
      Type the name of the listener port for this message-driven bean.
7. Save your changes to the deployment descriptor:
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.
8. Verify the archive files.
9. From the pop-up menu for the project, click **Deploy** to generate EJB deployment code.
10. Optional: Test your completed module on a WebSphere Application Server installation.
    Right-click a module, click **Run on Server**, then follow the instructions in the displayed wizard.

    Restriction: **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only. You cannot deploy remotely to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important: Use **Run on Server** for unit testing only. When an application is published remotely, the assembly tool overwrites the server configuration file for that server. Do not use on production servers.

What to do next

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that will run the application; for example, use the administrative console as described in Deploying and managing applications.

Chapter 44. Deploying OSGi applications

You deploy an OSGi application by adding an enterprise bundle archive (EBA) asset to a business-level application.

About this task

You can add an EBA asset to a business-level application by using the administrative console, or by using wsadmin commands.

The WebSphere Application Server administrative console and the OSGi Applications command-line console provide commands that you can use to explore or debug bundles running on an application server.

Procedure

- Deploy an OSGi application as a business-level application.

  Import an OSGi application as an enterprise bundle archive (EBA) asset, then add the asset to a business-level application by creating a composition unit. Optionally, add a composite bundle extension to the composition unit..
- Debug bundles at run time.

  You can use either the WebSphere Application Server administrative console or the wsadmin-based OSGi Applications command-line console to explore or debug the bundles associated with a specific OSGi application or shared bundle framework..

Deploying an OSGi application as a business-level application

Import an OSGi application as an enterprise bundle archive (EBA) asset, then add the asset to a business-level application by creating a composition unit. Optionally, add a composite bundle extension to the composition unit.

Before you begin

This topic assumes that you have already created an enterprise OSGi application packaged as an EBA file, for example as described in “Creating an OSGi application” on page 668. You might also have developed a composite bundle extension, as described in Extending a deployed OSGi application.

In addition to specifying the configuration information for the EBA asset through the following procedure, you can also change it later as described in Modifying the configuration of an OSGi composition unit. For example, if you update a bundle in an EBA asset, or replace a composite bundle extension, you might introduce a resource that requires additional configuration, such as a new or changed Blueprint resource reference, or security role mapping.

About this task

To deploy an OSGi application in WebSphere Application Server, you import your OSGi application (EBA file) as an asset and create an empty business-level application. You then add a composition unit to the business-level application. This composition unit consists of the new EBA asset plus configuration information for the context roots, virtual hosts, security role mappings, and web application or Blueprint resource bindings for your OSGi application. The composition unit can also include composite bundle extensions.

Note:

- An EBA file can be imported into only one asset.

- An EBA asset can be added to only one business-level application.
- One or more composite bundle extensions can be added to a composition unit.

A business-level application is scoped to cell scope, therefore only one instance of a given OSGi application can be deployed in a cell.

This topic describes the specific task of deploying an OSGi application and any composite bundle extensions as a business-level application. The more generalized task of creating any business-level application is described in Creating business-level applications.

Each step can be completed using either the administrative console or wsadmin commands. You can also create an empty business-level application or add a composition unit using programming.

Demonstration of this task (4 min) 

Procedure

1. Import the EBA file as an asset.

   Note: You can import the asset before or after you create the empty business-level application. An EBA file can be imported into only one asset.

   To import the EBA file using the administrative console, navigate to **Applications** > **New Application** > **New Asset**. For more information, see Importing assets.

   To import the EBA file using the **importAsset** command, enter (for example) the following command:

   ```
   AdminTask.importAsset(["-storageType", "FULL",
                          "-source", com.ibm.ws.eba.helloWorldService.eba])
   ```

   For more information, see the step “Import assets to your configuration” in topic Setting up business-level applications using wsadmin scripting.

   Notes:

   - When you import the EBA file as an asset, it is checked for any bundle dependencies. If the OSGi application has dependencies on bundles that are not included in the EBA file, the dependencies are resolved against any configured bundle repositories. Asset registration cannot complete unless all missing dependencies are available from configured bundle repositories.
   - When asset registration completes, if the asset requires bundles to be downloaded from bundle repositories, a warning message is displayed telling you to save your changes to the master configuration after completing the asset import. When you save changes, the missing dependencies are downloaded from the configured bundle repositories.
   - If the asset uses Java 2 security, the security permissions are displayed. This information comes from the `META-INF/permissions.perm` file for your application. For more information, see Java 2 security and OSGi Applications.
   - Do not add the asset to a business-level application until the bundle download has completed. You can view the download status of the bundles from the administrative console, or by calling the areAllDownloadsComplete () method of the BundleCacheManager MBean. See Interacting with the OSGi bundle cache.

2. Create an empty business-level application.

   To create an empty business-level application using the administrative console, navigate to **Applications** > **New Application** > **New Business Level Application**. For more information, see the step “Create an empty business-level application” in topic Creating business-level applications with the console.

   To create an empty business-level application using the **createEmptyBLA** command, enter (for example) the following command:

```
AdminTask.createEmptyBLA('[-name "helloWorld"
                           -description "helloWorld OSGi sample"]')
```

For more information, see the step “Create an empty business-level application” in topic Setting up business-level applications using wsadmin scripting.

To create an empty business-level application using programming, see Creating an empty business-level application using programming.

3. Add the EBA asset to the business-level application as a composition unit.

   An EBA asset can be added to only one business-level application. An OSGi composition unit consists of an EBA asset, (optionally) one or more composite bundle extensions, and some or all of the following configuration information:

   - Mappings from the composition unit to a target application server, web server, or cluster.
   - Configuration of the session manager, context roots or virtual hosts of the application.
   - Mappings from enterprise beans to JNDI names.
   - Bindings to any associated web applications or blueprint resource references.
   - Mappings from security roles to particular users or groups.

   To add the asset to the business-level application as a composition unit, use one of the following methods:

   - “Adding an EBA asset to a composition unit by using the administrative console.”
   - “Adding an EBA asset to a composition unit by using wsadmin commands” on page 2096.
   - Adding a composition unit using programming.

4. Optional: Add a composite bundle as an extension to the composition unit.

   After you import the enterprise bundle archive (EBA) file for your OSGi application as an asset, you can update versions of existing bundles but you cannot add extra bundles to the asset. However, after you have added the asset as a composition unit to a business-level application, you can extend the business-level application by adding one or more composite bundles to the composition unit.

5. Save your changes to the master configuration.

   If you are using wsadmin commands, enter the following command:

   ```
   AdminConfig.save()
   ```

What to do next

You are now ready to start your business-level application.

Adding an EBA asset to a composition unit by using the administrative console

Use the administrative console to add a composition unit that consists of a previously-imported EBA asset plus configuration information. The configuration information can include HTTP session management, context roots, virtual hosts, security roles, run-as roles, JNDI mappings for Session enterprise beans, JNDI mappings for EJB references, and web application or Blueprint resource reference bindings for your OSGi application.

Before you begin

You can add an EBA asset to a business-level application by using the administrative console as described in this topic, or by using the **addCompUnit** command as described in “Adding an EBA asset to a composition unit by using wsadmin commands” on page 2096.

An EBA asset can be added to only one business-level application. A business-level application is scoped to cell scope, therefore only one instance of a given OSGi application can be deployed in a cell.

This task makes the following assumptions:

- You have already imported the EBA file as an asset (as described in “Deploying an OSGi application as a business-level application” on page 2091), then saved your changes to the master configuration (which causes any bundle dependencies to be downloaded from configured bundle repositories).
- You have already defined the target virtual hosts. To check existing virtual hosts by using the administrative console, click **Environment** > **Virtual hosts**.
- You have already created the JCA authentication alias that you want to associate with each Blueprint resource reference. To check existing JCA authentication aliases using the administrative console, click **Security** > **Global security** > **[Authentication] Java Authentication and Authorization Service** > **J2C authentication data**.

About this task

An OSGi composition unit consists of an EBA asset, (optionally) one or more composite bundle extensions, and some or all of the following configuration information:

- Mappings from the composition unit to a target application server, web server, or cluster.
- Configuration of the session manager, context roots or virtual hosts of the application.
- Mappings from enterprise beans to JNDI names.
- Bindings to any associated web applications or blueprint resource references.
- Mappings from security roles to particular users or groups.

You use the Set options settings wizard to add a new composition unit to a business-level application. The contents of the asset determine the specific steps in the wizard. This topic describes the main elements that you configure when adding an EBA asset. For a general description of all the elements that the wizard might prompt you to configure, see Creating business-level applications with the console.

In addition to specifying the configuration information for the EBA asset through the following procedure, you can also change it later as described in Modifying the configuration of an OSGi composition unit. For example, if you update a bundle in an EBA asset, or replace a composite bundle extension, you might introduce a resource that requires additional configuration, such as a new or changed Blueprint resource reference, or security role mapping.

Procedure

1. Start the administrative console.
2. Add the previously-imported asset (the `.eba` file) as a deployed asset.
   a. If you have just created a new business-level application, the general properties for the business-level application are already displayed, including the option **[Deployed assets] Add** > **Add Asset**. Otherwise, navigate to **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > **[Deployed assets] Add** > **Add Asset**. A list of available assets is displayed.
   b. Select the asset to add, then click **Continue**. The Set options settings wizard is displayed.
3. Wizard step: Set options.

   Change the composition unit settings as needed, then click **Next**. For more information, see Set options settings.
4. Wizard step: Map composition unit to a target.

   Select the deployment target application server or web server. For more information, see Map target settings.
5. Wizard step: Map context root for web modules.

   Select a web application bundle (WAB) from the list, then enter the context root for the WAB. For example, `/sample`. For more information, see Context root for web modules [Settings].
6. Wizard step: Map virtual hosts for web modules.

The list of available WABs in this asset is displayed. For each WAB, you can change the associated virtual host by selecting a different one from the list. If you specify an existing virtual host in the `ibm-web-bnd.xml` or `.xmi` file for a WAB, the specified virtual host is set by default. Otherwise, the default virtual host setting is `default_host`. For more information, see Virtual hosts for web modules [Settings].

7. Wizard step: Map security roles to users or groups.

   Change the security mapping as needed. For more information, see Security role to user or group mapping [Settings].

8. Wizard step: Map RunAs roles to users

   You can map a specified user identity and password to a RunAs role. This mapping enables you to specify application-specific privileges for individual users, so that they can run specific tasks using another user identity. For more information, see RunAs roles for users [Collection].

9. Wizard step: Bind Blueprint resource references.

   The list of available Blueprint resource references in this asset is displayed. For each reference, you can optionally select an authentication alias from the list. Default authentication aliases (from `ibm-eba-bnd.xml` files) are offered only if they exist on every target server or cluster. For more information, see Blueprint resource references [Settings].

10. Wizard step: Map web module resource references to resources.

    The list of available web application resource references in this asset is displayed. That is, resources of type resource-ref (resource reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform. For each reference, specify the JNDI name under which the resource is known in the runtime environment. Optionally, set authentication properties and extended data source custom properties, which affect how the resource is accessed at run time. To specify the JNDI name mapping, either type the JNDI name into the box, or click **Browse...** then select the resource reference from the list of available resources. To modify the authentication method, or to set extended data source custom properties that apply to the database connection, select a single reference then click **Modify Resource Authentication Method...** or **Extended Properties...**. For more information, see Web module resource references [Settings].

11. Wizard step: Bind web module message destination references to administered objects.

    The list of available web application message destination and resource environment references in this asset is displayed. That is, resources of type message-destination-ref (message destination reference) or resource-env-ref (resource environment reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform. For each reference, specify the JNDI name under which the resource is known in the runtime environment. For more information, see Web module message destination references [Settings].

12. Wizard step: Provide EJB JNDI names

    For each Session enterprise bean in the composition unit, you can specify the JNDI name by which the enterprise bean is known in the runtime environment. For more information, see EJB JNDI names [Settings].

13. Wizard step: Map EJB References

    For each EJB reference that is defined in either an ejb-jar.xml file, a web.xml file, or an @EJB annotation in the composition unit, you can specify the JNDI name by which the EJB reference is known in the runtime environment. For more information, see EJB references [Settings].

14. Wizard step: Map EJB resource references to resources

    The list of available EJB resource references in this asset is displayed. That is, resources of type resource-ref (resource reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform. For each reference, specify the JNDI name under which the resource is known in the runtime environment. For more information, see EJB resource references [Settings].

15. Wizard step: Bind EJB message destination references to administered objects

    The list of available EJB message destination and resource environment references in this asset is displayed. That is, resources of type message-destination-ref (message destination reference) or

resource-env-ref (resource environment reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform. For each reference, specify the JNDI name under which the resource is known in the runtime environment. For more information, see EJB message destination references [Settings].

16. Wizard step: Bind listeners for message-driven beans

    For each message-driven bean (MDB) that is defined in either an ejb-jar.xml file or an @MessageDriven annotation in the composition unit, you can specify the settings necessary to bind an MDB listener to the MDB. By binding a listener to an MDB, you configure the association of the MDB with the JMS destination from which the MDB receives messages. For more information, see Listeners for message-driven beans [Settings].

17. Wizard step: Summary

    A summary of your selections is displayed. To complete the creation of the composition unit, click **Finish**. If there are settings that you want to change, click **Previous** to review the settings.

18. Save your changes to the master configuration.

Results

The product creates composition units for the application, module, or shared library relationships. The unit names are shown in lists of deployed assets on the settings page of your business-level application. To view the settings page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name***.

What to do next

Note: After you import the enterprise bundle archive (EBA) file for your OSGi application as an asset, you can update versions of existing bundles but you cannot add extra bundles to the asset. However, after you have added the asset as a composition unit to a business-level application, you can extend the business-level application by adding one or more composite bundles to the composition unit. See Adding or removing extensions for an OSGi composition unit.

You are now ready to start your business-level application.

Adding an EBA asset to a composition unit by using wsadmin commands

You can use the **addCompUnit** command and the **AdminConfig** commands to add a composition unit that consists of a previously-imported enterprise bundle archive (EBA) asset plus configuration information. The configuration information can include HTTP session management, context roots, virtual hosts, security roles, run-as roles, JNDI mappings for Session enterprise beans, JNDI mappings for EJB references, and web application or Blueprint resource reference bindings for your OSGi application.

Before you begin

You can add an EBA asset to a business-level application by using wsadmin commands as described in this topic, or by using the administrative console as described in “Adding an EBA asset to a composition unit by using the administrative console” on page 2093.

An EBA asset can be added to only one business-level application. A business-level application is scoped to cell scope, therefore only one instance of a given OSGi application can be deployed in a cell.

This task makes the following assumptions:

- You have already imported the EBA file as an asset (for example by using the **importAsset** command as described in step 2 of Setting up business-level applications using wsadmin scripting), then saved your changes to the master configuration (which causes any bundle dependencies to be downloaded from configured bundle repositories).

- You have already defined the target virtual hosts. To check existing virtual hosts using the wsadmin tool, issue the following jython command:

  ```
  print AdminConfig.list('VirtualHost')
  ```
- You have already created the JCA authentication alias that you want to associate with each Blueprint resource reference (for more information, see Configuring new Java 2 Connector authentication data entries using wsadmin).

About this task

An OSGi composition unit consists of an EBA asset, (optionally) one or more composite bundle extensions, and some or all of the following configuration information:

- Mappings from the composition unit to a target application server, web server, or cluster.
- Configuration of the session manager, context roots or virtual hosts of the application.
- Mappings from enterprise beans to JNDI names.
- Bindings to any associated web applications or blueprint resource references.
- Mappings from security roles to particular users or groups.

To create and configure all elements of the composition unit except the HTTP session manager, you use the **addCompUnit** command. To configure the HTTP session manager, you use the **AdminConfig** commands to configure the deployed object represented by the *appDeploy* variable. The composition unit must be created before the session management options can be applied to it, so you must run the **addCompUnit** command before you configure the HTTP session manager.

In addition to specifying the configuration information for the EBA asset through the following procedure, you can also change it later as described in Modifying the configuration of an OSGi composition unit by using wsadmin commands. For example, if you update a bundle in an EBA asset, or replace a composite bundle extension, you might introduce a resource that requires additional configuration, such as a new or changed Blueprint resource reference, or security role mapping.

Procedure

1. Create and configure all elements of the composition unit except the HTTP session manager.

   Each of the following substeps describes the syntax for adding a single element to the composition unit. However, to create the composition unit you run the **addCompUnit** command only once. Therefore, when you run the command, you must combine these elements together. An example of the combined syntax is given after the separate substeps.

   For several of the elements, the values you specify include bundle identifiers. If your EBA asset includes or references composite bundles, the command syntax is slightly different. For clarity, the differences for composite bundles are described, step by step, in a linked topic.

   a. Add the previously-imported asset (the `.eba` file) as a deployed asset.

      The parameters for this aspect of the command are as follows:

      -blaID
      Specifies the configuration ID of the business-level application.

      -cuSourceID
      Specifies the ID of the EBA asset that is being added to the business-level application.

      -CUOptions
      Specifies the following additional properties for the composition unit.
      - parentBLA
      - backingID
      - name
      - description

- startingWeight
- startedOnDistributed
- restartBehaviorOnUpdate

-MapTargets
Specifies additional properties for the composition unit target mapping. That is, it specifies the deployable unit URI (which, for an EBA asset, is `ebaDeploymentUnit`) and the target node and server, or the target cluster. To add an additional target, you use the plus sign character (+) as a prefix.

-ActivationPlanOptions
Specifies additional properties for the composition unit activation plan. That is, it specifies the deployable unit URI and a list of runtime components. For an EBA asset, use default values as shown in the example that follows.

If the target is one cluster, the Jython syntax for this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=bla_name
  -cuSourceID WebSphere:assetname=asset_name.eba
  -CUOptions [
    [WebSphere:blaname=bla_name.eba
    WebSphere:assetname=asset_name.eba
    cu_name "optional_cu_description" 1 false DEFAULT]]
  -MapTargets [[ebaDeploymentUnit WebSphere:cluster=cluster_name]]
  -ActivationPlanOptions [[default ""]]
  ...
]')
```

For example:

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=helloWorldService
  -cuSourceID WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
  -CUOptions [
    [WebSphere:blaname=helloWorldService.eba
    WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
    com.ibm.ws.eba.helloWorldService_0001.eba "" 1 false DEFAULT]]
  -MapTargets [[ebaDeploymentUnit WebSphere:cluster=cluster1]]
  -ActivationPlanOptions [[default ""]]
  ...
]')
```

If the target is two servers, the Jython syntax for this aspect of the command is as follows::

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=bla_name
  -cuSourceID WebSphere:assetname=asset_name.eba
  -CUOptions [
    [WebSphere:blaname=bla_name.eba
    WebSphere:assetname=asset_name.eba
    cu_name "optional_cu_description" 1 false DEFAULT]]
  -MapTargets [
    [ebaDeploymentUnit WebSphere:node=node_name,server=server_name+
                       WebSphere:node=node2_name,server=server2_name]]
  -ActivationPlanOptions [[default ""]]
  ...
]')
```

For example:

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=helloWorldService
  -cuSourceID WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
  -CUOptions [
    [WebSphere:blaname=helloWorldService.eba
    WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
    com.ibm.ws.eba.helloWorldService_0001.eba "" 1 false DEFAULT]]
```

```
  -MapTargets [[ebaDeploymentUnit WebSphere:node=node01,server=server1+
                                  WebSphere:node=node01,server=web1]]
  -ActivationPlanOptions [[default ""]]
  ...
]')
```

b. Map context root for web modules.

Context roots determine where the web pages of a particular web application bundle (WAB) are found at run time. The context root that you specify here is combined with the defined server mapping to compose the full URL that you enter to access the pages of the WAB. For example, if the application server default host is `www.example.com:8080` and the context root of the WAB is `/sample`, the web pages are available at `www.example.com:8080/sample`.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Map context root for web modules in composite bundles.

The list of bundles under the **ContextRootStep** must contain all the WABs contained in the OSGi application.

```
AdminTask.addCompUnit('[
  ...
  -ContextRootStep [
    [bundle_symbolic_name_1 bundle_version_1 context_root_1]
    [bundle_symbolic_name_2 bundle_version_2 context_root_2]]
  ...
]')
```

For example, for an EBA file that contains two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `/hello/web`, and `com.ibm.ws.eba.helloWorldService.withContextRoot` at version 0.9.0, which is to be mapped to `/hello/service`), this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -ContextRootStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 "/hello/web"]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "/hello/service"]]
  ...
]')
```

c. Bind listeners for message-driven beans

For each message-driven bean (MDB) that is defined in either an ejb-jar.xml file or an @MessageDriven annotation in the composition unit, you can specify the settings necessary to bind an MDB listener to the MDB. By binding a listener to an MDB, you configure the association of the MDB with the JMS destination from which the MDB receives messages.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Bind listeners for message-driven beans in composite bundles

```
AdminTask.addCompUnit('[
  ...
  -MDBBindingsStep [
    [bundle_symbolic_name_1 bundle_version_1 uri_1
       activation_spec_1 destination_jndi_name_1 authentication_alias_1]
    [bundle_symbolic_name_2 bundle_version_2 uri_2
       activation_spec_2 destination_jndi_name_2 authentication_alias_2]]
  ...
]')
```

In the following example, an EBA file contains two EJB bundles, `com.ibm.ws.eba.currencyService` at version 1.0.0, and `com.ibm.ws.eba.accountService` at version 0.9.0. The currencyService bundle contains a message-driven bean called `ExchangeRateMDB`, bound to an activation specification with

a JNDI name of eis/ExchangeRate_Act_Spec; the destination JNDI name that is defined in the activation specification is overridden by a destination whose JNDI name is jms/ExchangeRateQueue, and the authentication alias that is defined in the activation specification is overridden by an authentication alias called ExchangeRate_Auth_Alias. The accountService bundle contains an MDB called CustomerDetailsMDB, bound to an activation specification with a JNDI name of eis/CustomerDetails_Act_Spec; the destination JNDI name that is defined in the activation specification is overridden by a destination whose JNDI name is jms/CustomerDetailsQueue, and the authentication alias that is defined in the activation specification is overridden by an authentication alias called CustomerDetails_Auth_Alias.

```
AdminTask.addCompUnit('[
  ...
  -MDBBindingsStep [
    [com.ibm.ws.eba.currencyService 1.0.0 META-INF/ejb-jar.xml/ExchangeRateMDB
      eis/ExchangeRate_Act_Spec jms/ExchangeRateQueue ExchangeRate_Auth_Alias]
    [com.ibm.ws.eba.accountService 0.9.0 META-INF/ejb-jar.xml/CustomerDetailsMDB
      eis/CustomerDetails_Act_Spec jms/CustomerDetailsQueue CustomerDetails_Auth_Alias]]
  ...
]')
```

d. Provide EJB JNDI names

For each Session enterprise bean in the composition unit, you can specify the JNDI name by which the enterprise bean is known in the runtime environment.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Provide EJB JNDI names in composite bundles

```
AdminTask.addCompUnit('[
  ...
  -EJBMappingsStep [
    [bundle_symbolic_name_1 bundle_version_1 ejb_name_1
       ejb_interface_1 ejb_iterface_type_1 jndi_name_1]
    [bundle_symbolic_name_2 bundle_version_2 ejb_name_2
       ejb_interface_2 ejb_iterface_type_2 jndi_name_2]]
  ...
]')
```

In the following example, an EBA file contains two EJB bundles, com.ibm.ws.eba.currencyService at version 1.0.0, and com.ibm.ws.eba.accountService at version 0.9.0. The currencyService bundle contains an enterprise bean called ExchangeRate_ejb, with a Local interface called com.ibm.ws.eba.ejb.ExchangeRate, that is mapped to a JNDI name of ejb/ExchangeRate. The accountService bundle contains an enterprise bean called CustomerDetails_ejb, with a Remote interface called com.ibm.ws.eba.ejb.CustomerDetails, that is mapped to a JNDI name of ejb/CustomerDetails.

```
AdminTask.addCompUnit('[
  ...
  -EJBMappingsStep [
    [com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate_ejb
      com.ibm.ws.eba.ejb.ExchangeRate Local ejb/ExchangeRate]
    [com.ibm.ws.eba.accountService 0.9.0 CustomerDetails_ejb
      com.ibm.ws.eba.ejb.CustomerDetails Remote ejb/CustomerDetails]]
  ...
]')
```

e. Map EJB references

For each EJB reference that is defined in either an ejb-jar.xml file, a web.xml file, or an @EJB annotation in the composition unit, you can specify the JNDI name by which the EJB reference is known in the runtime environment.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Map EJB references in composite bundles

```
AdminTask.addCompUnit('[
  ...
  -EJBRefStep [
    [bundle_symbolic_name_1 bundle_version_1 uri_1
       ejb_reference_name_1 business_interface_1 jndi_name_1]
    [bundle_symbolic_name_2 bundle_version_2 uri_2
       ejb_reference_name_2 business_interface_2 jndi_name_2]]
  ...
]')
```

The *uri* parameter specifies the location where the EJB reference is defined.

In the following example, an EBA file contains two bundles, `com.ibm.ws.eba.currencyService` at version 1.0.0, and `com.ibm.ws.eba.accountService` at version 0.9.0. The currencyService bundle contains an EJB reference called `ExchangeRate`, from the `CurrencyExchange` enterprise bean, defined in `META-INF/ejb-jar.xml`, that is mapped to a JNDI name of `ejb:ExchangeRate`. The accountService bundle contains an EJB reference called `CustomerDetails`, defined in `web.xml`, that is mapped to a JNDI name of `ejb:CustomerDetails`.

```
AdminTask.addCompUnit('[
  ...
  -EJBRefStep [
    [com.ibm.ws.eba.currencyService 1.0.0 META-INF/ejb-jar.xml/CurrencyExchange
      ExchangeRate com.ibm.ws.eba.ejb.ExchangeRate ejb:ExchangeRate]
    [com.ibm.ws.eba.accountService 0.9.0 WEB-INF/web.xml
      CustomerDetails com.ibm.ws.eba.ejb.CustomerDetails ejb:CustomerDetails]]
  ...
]')
```

f. Map EJB resource references to resources

Binding a resource reference maps a resource dependency of an enterprise bean to an actual resource available in the server runtime environment. At a minimum, you can be achieve this mapping by specifying the JNDI name under which the resource reference is known in the runtime environment. By default, the JNDI name is retrieved from pre-existing bindings, or set to the value of the mapped-name specified in the resource reference definition. Use this option to bind resources of type resource-ref (resource reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Map EJB resource references to resources in composite bundles

```
AdminTask.addCompUnit('[
  ...
  -EJBResourceRefs [
    [
    bundle_symbolic_name
    bundle_version
    ejb_name
    resource_reference_id
    resource_type
    target_jndi_name
    resource_authentication_method
    mapping_properties
    extended_properties
    ]]
  ...
]')
```

The *mapping_properties* parameter defines arbitrary name and value pairs for extended data source properties, in the following format (one continuous string):

```
WebSphere:name=property_name1,value=property_value1,description=property_description1
+WebSphere:name=property_name2,value=property_value2,description=property_description2
+ ...
```

The *extended_properties* parameter defines extended data source custom properties in the following format (one continuous string):

```
property_name1=property_value1+property_name2=property_value2+ ...
```

For example:

```
AdminTask.addCompUnit('[
  ...
  -EJBResourceRefs [
    [com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate
     dataSource1 javax.sql.DataSource ref/ds1 ClientContainer
     "WebSphere:name=mprop1,value=val1,description=desc1"
     "exprop1=expropval1+exprop2=expropval2"]
    [com.ibm.ws.eba.accountService 0.9.0 CustomerDetails
     dataSource2 javax.sql.DataSource ref/ds2 WSLogin "" ""]]
  ...
]')
```

g. Bind EJB message destination references to administered objects.

Binding a message destination reference or resource environment reference maps a resource dependency of an enterprise bean to an actual resource available in the server runtime environment. At a minimum, you can achieve this mapping by specifying the JNDI name under which the message destination reference or resource environment reference is known in the runtime environment. By default, the JNDI name is retrieved from pre-existing bindings, or set to the value of the mapped-name specified in the message destination reference definition. Use this option to bind resources of type message-destination-ref (message destination reference) or resource-env-ref (resource environment reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Bind EJB message destination references to administered objects in composite bundles.

```
AdminTask.addCompUnit('[
  ...
  -EJBMsgDestRefs [
    [
    bundle_symbolic_name
    bundle_version
    ejb_name
    resource_reference_id
    resource_type
    target_jndi_name
    ]]
  ...
]')
```

For example:

```
AdminTask.addCompUnit('[
  ...
  -EJBMsgDestRefs [
    [com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate
     jms/myQ javax.jms.Queue jms/workQ]
    [com.ibm.ws.eba.accountService 0.9.0 CustomerDetails
     jms/myT javax.jms.Topic jms/notificationTopic]]
  ...
]')
```

h. Map virtual hosts for web modules.

You use a virtual host to associate a unique port with a web application. The aliases of a virtual host identify the port numbers defined for that virtual host. A port number specified in a virtual host

alias is used in the URL that is used to access artifacts such as servlets and JavaServer Page (JSP) files in a web application. For example, the alias `myhost:8080` is the *host_name:port_number* portion of the URL `http://myhost:8080/sample`.

Each WAB that is contained in a deployed asset must be mapped to a virtual host. WABs can be installed on the same virtual host, or dispersed among several virtual hosts.

If you specify an existing virtual host in the `ibm-web-bnd.xml` or `.xmi` file for a WAB, the specified virtual host is set by default. Otherwise, the default virtual host setting is `default_host`, which provides several port numbers through its aliases:

80 An internal, insecure port used when no port number is specified
9080 An internal port
9443 An external, secure port

Unless you want to isolate your WAB from other WABs or resources on the same node, `default_host` is a suitable virtual host. In addition to `default_host`, WebSphere Application Server provides `admin_host`, which is the virtual host for the administrative console system application. `admin_host` is on port 9060. Its secure port is 9043. Do not select `admin_host` unless the WAB relates to system administration.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Map virtual hosts for web modules in composite bundles.

```
AdminTask.addCompUnit('[
  ...
  -VirtualHostMappingStep [
    [bundle_symbolic_name_1 bundle_version_1
    web_module_name_1 virtual_host_1]
    [bundle_symbolic_name_2 bundle_version_2
     web_module_name_2 virtual_host_2]]
  ...
]')
```

For example, for an EBA file containing two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `default_host`, and `com.ibm.ws.eba.helloWorldService.withContextRoot` at version 0.9.0, which is to be mapped to `test_host`), this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0
    "HelloWorld second service" test_host]]
  ...
]')
```

i. Map security roles to users or groups.

The Jython syntax for this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -MapRolesToUsersStep [
    [role_name everyone?
    all_authenticated_in_realm?
    usernames groups]]
  ...
]')
```

Key:

- *role_name* is a role name defined in the application.
- *everyone?* is set to `Yes` or `No`, to specify whether everyone is in the role.

- *all_authenticated_in_realm?* is set to `Yes` or `No`, to specify whether all authenticated users can access the application realm.
- *usernames* is a list of WebSphere Application Server user names, separated by the "|" character.
- *groups* is a list of WebSphere Application Server groups, separated by the "|" character.

Note: For *usernames*, and *groups*, the empty string "" means "use the default or existing value". The default value is usually that no users or groups are bound to the role. However, when an application contains an `ibm-application-bnd.xmi` file, the default value for *usernames* is obtained from this file. If you are deploying an application that contains an `ibm-application-bnd.xmi` file, and you want to remove the bound users, specify just the "|" character (which is the separator for multiple user names). This setting explicitly specifies "no users", and therefore guarantees that no users are bound to the role.

For example:

```
AdminTask.addCompUnit('[
  ...
  -MapRolesToUsersStep [
    [ROLE1 No Yes "" ""]
    [ROLE2 No No WABTestUser1 ""]
    [ROLE3 No No "" WABTestGroup1]
    [ROLE4 Yes No "" ""]]
  ...
]')
```

For more information about the -MapRolesToUsersStep option, see the information for the $AdminApp install command “MapRolesToUsers” option. This is the equivalent option for Java EE applications. For more general information, see Security role to user or group mapping.

j. Map RunAs roles to users

You can map a specified user identity and password to a RunAs role. This mapping enables you to specify application-specific privileges for individual users, so that they can run specific tasks using another user identity. The Jython syntax for this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -MapRunAsRolesToUsersStep [
    [role_name user_name password]]
]')
```

For example:

```
AdminTask.addCompUnit('[
  ...
  -MapRunAsRolesToUsersStep [
    [Role1 User1 password1]
    [AdminRole User3 password3]]
]')
```

For more information about the -MapRunAsRolesToUsers option, see the information for the $AdminApp install command “MapRunAsRolesToUsers” option. This is the equivalent option for Java EE applications. For more general information, see Map RunAs roles to users.

k. Add authentication aliases to Blueprint resource references.

Blueprint components can access WebSphere Application Server resource references. Each reference is declared in a Blueprint XML file, and can be secured using a Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) authentication alias. Each bundle in an OSGi application can contain any number of resource reference declarations in its various Blueprint XML files.

When you secure resource references, those resource references can be bound only to JCA authentication aliases that exist on every server or cluster on which the application is deployed. An OSGi application can be deployed to multiple servers and clusters that are in the same security

domain. Therefore, each JCA authentication alias must exist in either the security domain of the target servers and clusters, or the global security domain.

You must declare the resource references in the Blueprint XML file. For example:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:rr="http://www.ibm.com/appserver/schemas/8.0/blueprint/resourcereference">
  <!-- Other Blueprint declarations ... -->

  <rr:resource-reference id="resourceRef"
          interface="javax.sql.DataSource"
          filter="(osgi.jndi.service.name=jdbc/Account)">
    <rr:res-auth>Container</rr:res-auth>
    <rr:res-sharing-scope>Shareable</rr:res-sharing-scope>
  </rr:resource-reference>
</blueprint>
```

This declaration includes the resource reference ID (for example `resourceRef`), the service filter (for example `jdbc/Account`), the authentication type (for example `Container`), and the sharing setting (for example `Shareable`).

The Blueprint resource references to authentication alias bindings for each bundle are stored in a file `ibm-eba-bnd.xml` in the `META-INF` directory of that bundle. If an OSGi application contains any of these files when it is deployed as an asset, these files provide the default authentication alias values that are used when binding the resource references. For example:

```
<eba-bnd>
  <resource-ref>
    <jndi-name>jdbc/Acount</jndi-name>
    <authentication-alias>Alias1</authentication-alias>
    <interface>javax.sql.DataSource</interface>
    <authentication>Container</authentication>
    <sharing-scope>Shareable</sharing-scope>
    <id>resourceRef</id>
  </resource-ref>
</eba-bnd>
```

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Add authentication aliases to Blueprint resource references in composite bundles.

```
AdminTask.addCompUnit('[
  ...
  -BlueprintResourceRefBindingStep [
    [
    bundle_symbolic_name
    bundle_version
    blueprint_resource_reference_id
    interface_name
    jndi_name
    authentication_type
    sharing_setting
    authentication_alias_name
    ]]
  ...
]')
```

Note: The value for *jndi_name* must match the JNDI name that you declare in the **filter** attribute of the resource reference element in the Blueprint XML file.

For example, for an EBA file that contains a bundle `com.ibm.ws.eba.helloWorldService.properties.bundle.jar` at Version 1.0.0, which is to be bound to authentication alias `alias1`, the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -BlueprintResourceRefBindingStep[
```

```
    [com.ibm.ws.eba.helloWorldService.properties.bundle 1.0.0 resourceRef
    javax.sql.DataSource jdbc/Account Container Shareable alias1]]
  ...
]')
```

l. Bind web module message destination references to administered objects.

Binding a resource reference maps a resource dependency of the web application to an actual resource available in the server runtime environment. At a minimum, you can achieve this mapping by specifying the JNDI name under which the resource is known in the runtime environment. By default, the JNDI name is the resource ID that you specified in the `web.xml` file during development of the web application bundle (WAB). Use this option to bind resources of type message-destination-ref (message destination reference) or resource-env-ref (resource environment reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Bind web module message destination references to administered objects in composite bundles.

```
AdminTask.addCompUnit('[
  ...
  -WebModuleMsgDestRefs [
    [
    bundle_symbolic_name
    bundle_version
    resource_reference_id
    resource_type
    target_jndi_name
    ]]
  ...
]')
```

For example:

```
AdminTask.addCompUnit('[
  ...
  -WebModuleMsgDestRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myQ javax.jms.Queue
    jms/workQ]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myT javax.jms.Topic
    jms/notificationTopic]]
  ...
]')
```

m. Map web module resource references to resources.

Binding a resource reference maps a resource dependency of the web application to an actual resource available in the server runtime environment. At a minimum, you can achieve this mapping by specifying the JNDI name under which the resource is known in the runtime environment. By default, the JNDI name is the resource ID that you specified in the `web.xml` file during development of the web application bundle (WAB). Use this option to bind resources of type resource-ref (resource reference), as defined in the Java specification JSR-250: Common Annotations for the Java Platform.

The Jython syntax for this aspect of the command is as follows.

Note: For composite bundles, the syntax is slightly different. See Step: Map web module resource references to resources in composite bundles.

```
AdminTask.addCompUnit('[
  ...
  -WebModuleResourceRefs [
    [
    bundle_symbolic_name
```

```
      bundle_version
      resource_reference_id
      resource_type
      target_jndi_name
      login_configuration
      login_properties
      extended_properties
      ]]
    ...
]')
```

For example:

```
AdminTask.addCompUnit('[
  ...
  -WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
  ...
]')
```

Note: If you use multiple extended properties, the jython syntax is `"extprop1=extval1,extprop2=extval2"`.

In the following example, the jython syntax from the previous individual substeps is combined so that, by running the **addCompUnit** command once only, a composition unit is created and added to a business-level application.

Note: If your EBA asset includes or references composite bundles, the command syntax is slightly different. For an equivalent example, see Example: Using the **addCompUnit** command to add an EBA asset that includes composite bundles.

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=helloWorldService
  -cuSourceID WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
  -CUOptions [
    [WebSphere:blaname=helloWorldService.eba
    WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
    com.ibm.ws.eba.helloWorldService_0001.eba "" 1 false DEFAULT]]
  -MapTargets [[ebaDeploymentUnit WebSphere:cluster=cluster1]]
  -ActivationPlanOptions [[default ""]]
  -ContextRootStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 "/hello/web"]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0 "/hello/service"]]
  -EJBMappingsStep [
    [com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate_ejb
      com.ibm.ws.eba.ejb.ExchangeRate Local ejb/ExchangeRate]
    [com.ibm.ws.eba.accountService 0.9.0 CustomerDetails_ejb
      com.ibm.ws.eba.ejb.CustomerDetails Remote ejb/CustomerDetails]]
  -EJBRefStep [
    [com.ibm.ws.eba.currencyService 1.0.0 META-INF/ejb-jar.xml/CurrencyExchange
      ExchangeRate com.ibm.ws.eba.ejb.ExchangeRate ejb:ExchangeRate]
    [com.ibm.ws.eba.accountService 0.9.0 WEB-INF/web.xml
      CustomerDetails com.ibm.ws.eba.ejb.CustomerDetails ejb:CustomerDetails]]
  -EJBResourceRefs [
    [com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate
     dataSource1 javax.sql.DataSource ref/ds1 ClientContainer
     "WebSphere:name=mprop1,value=val1,description=desc1"
     "exprop1=expropval1+exprop2=expropval2"]
    [com.ibm.ws.eba.accountService 0.9.0 CustomerDetails
     dataSource2 javax.sql.DataSource ref/ds2 WSLogin "" ""]]
  -EJBMsgDestRefs [
```

```
    [com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate
     jms/myQ javax.jms.Queue jms/workQ]
    [com.ibm.ws.eba.accountService 0.9.0 CustomerDetails
     jms/myT javax.jms.Topic jms/notificationTopic]]
  -VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0
    "HelloWorld second service" test_host]]
  -MapRolesToUsersStep [
    [ROLE1 No Yes "" ""]
    [ROLE2 No No WABTestUser1 ""]
    [ROLE3 No No "" WABTestGroup1]
    [ROLE4 Yes No "" ""]]
  -MapRunAsRolesToUsersStep [
    [Role1 User1 password1]
    [AdminRole User3 password3]]
  -BlueprintResourceRefBindingStep[
    [com.ibm.ws.eba.helloWorldService.properties.bundle 1.0.0 resourceRef
    javax.sql.DataSource jdbc/Account
    Container Shareable alias1]]
  -WebModuleMsgDestRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myQ javax.jms.Queue
    jms/workQ]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0
    jms/myT javax.jms.Topic
    jms/notificationTopic]]
  -WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldService.web 1.0.0 jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
]')
```

2. Configure the HTTP session manager.

   To configure the HTTP session manager, you use the **AdminConfig** commands to configure the deployed object represented by the *appDeploy* variable. Session management for OSGi applications is configured in the same way as for enterprise applications, except for a minor difference in syntax when getting the deployed object.

   a. Get the deployed object.

      Use the instructions given in Configuring applications for session management using scripting. Note that, for enterprise applications, you use the following two line script:

      ```
      deployments = AdminConfig.getid('/Deployment:myApp/')
      appDeploy = AdminConfig.showAttribute(deployments, 'deployedObject')
      ```

      For OSGi applications, the equivalent script is the following single line:

      ```
      appDeploy = AdminTask.getOSGiApplicationDeployedObject('-cuName cu_name')
      ```

      where *cu_name* is the name of the composition unit. For example:

      ```
      appDeploy = AdminTask.getOSGiApplicationDeployedObject('
        -cuName com.ibm.ws.eba.helloWorldService_0001.eba')
      ```

   b. Create the session management options.

      Use the instructions given in Configuring applications for session management using scripting. The command usage for creating the session management options is exactly the same for enterprise applications and OSGi applications.

What to do next

After using these commands, save your changes to the master configuration by using the following command:

```
AdminConfig.save()
```

Note: After you import the enterprise bundle archive (EBA) file for your OSGi application as an asset, you can update versions of existing bundles but you cannot add extra bundles to the asset. However, after you have added the asset as a composition unit to a business-level application, you can extend the business-level application by adding one or more composite bundles to the composition unit. See Adding or removing extensions for an OSGi composition unit using wsadmin commands.

You are now ready to start your business-level application.

Adding an EBA asset that includes composite bundles by using the addCompUnit command

You can use the **addCompUnit** command to add a composition unit that contains a previously-imported enterprise bundle archive (EBA) asset plus configuration information. If the EBA asset includes composite bundles, the command syntax is slightly different.

Before you begin

For a full description of how you specify this configuration information, see “Adding an EBA asset to a composition unit by using wsadmin commands” on page 2096. When you work through that task, each step where the syntax is different for composite bundles is linked to an equivalent step in this task.

About this task

An OSGi composition unit includes an EBA asset and some or all of the following configuration information:

- Mappings from the composition unit to a target application server, web server, or cluster.
- Configuration of the session manager, context roots or virtual hosts of the application.
- Mappings from enterprise beans to JNDI names.
- Bindings to any associated web applications or blueprint resource references.
- Mappings from security roles to particular users or groups.

For several of the elements, the values you specify include bundle identifiers. If your EBA asset includes or references composite bundles, the command syntax is slightly different. The differences for composite bundles are described in the following steps.

Procedure

- Map context root for web modules in composite bundles.

  The Jython syntax for this aspect of the command is as follows:

  ```
  AdminTask.addCompUnit('[
    ...
    -ContextRootStep [
      [bundle_symbolic_name_1 bundle_version_1 context_root_1]
      [bundle_symbolic_name_2 bundle_version_2 context_root_2]]
    ...
  ]')
  ```

  For composite bundles, the bundle symbolic name has the following syntax:

  ```
  CBA.symbolic.name_CBA.bundle.version/WAB.symbolic.name
  ```

  For example, for a composite bundle `com.ibm.ws.eba.helloWorldCBA` at version 1.0.0 that contains two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `/hello/web`,

and com.ibm.ws.eba.helloWorldService.withContextRoot at version 0.9.0, which is to be mapped to /hello/service), this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -ContextRootStep [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web 1.0.0
    "/hello/web"]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0
    "/hello/service"]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Map context root for web modules.

- Bind listeners for message-driven beans in composite bundles

  The Jython syntax for this aspect of the command is as follows.

```
AdminTask.addCompUnit('[
  ...
  -MDBBindingsStep [
    [bundle_symbolic_name_1 bundle_version_1 uri_1
       activation_spec_1 destination_jndi_name_1 authentication_alias_1]
    [bundle_symbolic_name_2 bundle_version_2 uri_2
       activation_spec_2 destination_jndi_name_2 authentication_alias_2]]
  ...
]')
```

  For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/EJBbundle.symbolic.name
```

  In the following example, a composite bundle com.ibm.ws.eba.financeCBA, at version 1.0.0, contains two EJB bundles, com.ibm.ws.eba.currencyService at version 1.0.0, and com.ibm.ws.eba.accountService at version 0.9.0. The currencyService bundle contains a message-driven bean called ExchangeRateMDB, bound to an activation specification with a JNDI name of eis/ExchangeRate_Act_Spec; the destination JNDI name that is defined in the activation specification is overridden by a destination whose JNDI name is jms/ExchangeRateQueue, and the authentication alias that is defined in the activation specification is overridden by an authentication alias called ExchangeRate_Auth_Alias. The accountService bundle contains an MDB called CustomerDetailsMDB, bound to an activation specification with a JNDI name of eis/CustomerDetails_Act_Spec; the destination JNDI name that is defined in the activation specification is overridden by a destination whose JNDI name is jms/CustomerDetailsQueue, and the authentication alias that is defined in the activation specification is overridden by an authentication alias called CustomerDetails_Auth_Alias.

```
AdminTask.addCompUnit('[
  ...
  -MDBBindingsStep [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
     META-INF/ejb-jar.xml/ExchangeRateMDB eis/ExchangeRate_Act_Spec
     jms/ExchangeRateQueue ExchangeRate_Auth_Alias]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
     META-INF/ejb-jar.xml/CustomerDetailsMDB eis/CustomerDetails_Act_Spec
     jms/CustomerDetailsQueue CustomerDetails_Auth_Alias]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Bind listeners for message-driven beans

- Provide EJB JNDI names in composite bundles

  The Jython syntax for this aspect of the command is as follows.

```
AdminTask.addCompUnit('[
  ...
  -EJBMappingsStep [
    [bundle_symbolic_name_1 bundle_version_1 ejb_name_1
       ejb_interface_1 ejb_interface_type_1 jndi_name_1]
    [bundle_symbolic_name_2 bundle_version_2 ejb_name_2
       ejb_interface_2 ejb_interface_type_2 jndi_name_2]]
  ...
]')
```

For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/EJBbundle.symbolic.name
```

In the following example, a composite bundle `com.ibm.ws.eba.financeCBA`, at version 1.0.0, contains two EJB bundles, `com.ibm.ws.eba.currencyService` at version 1.0.0, and `com.ibm.ws.eba.accountService` at version 0.9.0. The currencyService bundle contains an enterprise bean called `ExchangeRate_ejb`, with a Local interface called `com.ibm.ws.eba.ejb.ExchangeRate`, that is mapped to a JNDI name of `ejb/ExchangeRate`. The accountService bundle contains an enterprise bean called `CustomerDetails_ejb`, with a Remote interface called `com.ibm.ws.eba.ejb.CustomerDetails`, that is mapped to a JNDI name of `ejb/CustomerDetails`.

```
AdminTask.addCompUnit('[
  ...
  -EJBMappingsStep [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate_ejb
      com.ibm.ws.eba.ejb.ExchangeRate Local ejb/ExchangeRate]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0 CustomerDetails_ejb
      com.ibm.ws.eba.ejb.CustomerDetails Remote ejb/CustomerDetails]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Provide EJB JNDI names

- Map EJB references in composite bundles

  The Jython syntax for this aspect of the command is as follows.

```
AdminTask.addCompUnit('[
  ...
  -EJBRefStep [
    [bundle_symbolic_name_1 bundle_version_1 uri_1
       ejb_reference_name_1 business_interface_1 jndi_name_1]
    [bundle_symbolic_name_2 bundle_version_2 uri_2
       ejb_reference_name_2 business_interface_2 jndi_name_2]]
  ...
]')
```

  The *uri* parameter specifies the location where the EJB reference is defined.

  For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/EJBbundle.symbolic.name
```

  In the following example, a composite bundle `com.ibm.ws.eba.financeCBA`, at version 1.0.0, contains two bundles, `com.ibm.ws.eba.currencyService` at version 1.0.0, and `com.ibm.ws.eba.accountService` at version 0.9.0. The currencyService bundle contains an EJB reference called `ExchangeRate`, from the `CurrencyExchange` enterprise bean, defined in `META-INF/ejb-jar.xml` that is mapped to a JNDI name of `ejb:ExchangeRate`. The accountService bundle contains an EJB reference called `CustomerDetails`, defined in `web.xml`, that is mapped to a JNDI name of `ejb:CustomerDetails`.

```
AdminTask.addCompUnit('[
  ...
  -EJBRefStep [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
```

```
      META-INF/ejb-jar.xml/CurrencyExchange ExchangeRate com.ibm.ws.eba.ejb.ExchangeRate
      ejb:ExchangeRate]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
      WEB-INF/web.xml CustomerDetails com.ibm.ws.eba.ejb.CustomerDetails ejb:CustomerDetails]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Map EJB references

- Map EJB resource references to resources in composite bundles

  The Jython syntax for this aspect of the command is as follows

```
AdminTask.addCompUnit('[
  ...
  -EJBResourceRefs [
    [
    bundle_symbolic_name
    bundle_version
    ejb_name
    resource_reference_id
    resource_type
    target_jndi_name
    resource_authentication_method
    mapping_properties
    extended_properties
    ]]
  ...
]')
```

  The *mapping_properties* parameter defines arbitrary name and value pairs for extended data source properties, in the following format (one continuous string):

```
WebSphere:name=property_name1,value=property_value1,description=property_description1
+WebSphere:name=property_name2,value=property_value2,description=property_description2
+ ...
```

  The *extended_properties* parameter defines extended data source custom properties in the following format (one continuous string):

```
property_name1=property_value1+property_name2=property_value2+ ...
```

  For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/EJBBundle.symbolic.name
```

  For example:

```
AdminTask.addCompUnit('[
  ...
  -EJBResourceRefs [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
     ExchangeRate dataSource1 javax.sql.DataSource ref/ds1 ClientContainer
     "WebSphere:name=mprop1,value=val1,description=desc1"
     "exprop1=expropval1+exprop2=expropval2"]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
     CustomerDetails dataSource2 javax.sql.DataSource ref/ds2 WSLogin "" ""]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Map EJB resource references to resources

- Bind EJB message destination references to administered objects in composite bundles.

  The Jython syntax for this aspect of the command is as follows

```
AdminTask.addCompUnit('[
  ...
  -EJBMsgDestRefs [
```

```
    [
    bundle_symbolic_name
    bundle_version
    ejb_name
    resource_reference_id
    resource_type
    target_jndi_name
    ]]
  ...
]')
```

For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/EJBBundle.symbolic.name
```

For example:

```
AdminTask.addCompUnit('[
  ...
  -EJBMsgDestRefs [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
     ExchangeRate jms/myQ javax.jms.Queue jms/workQ]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
     CustomerDetails jms/myT javax.jms.Topic jms/notificationTopic]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Bind EJB message destination references to administered objects.

- Map virtual hosts for web modules in composite bundles.

  The Jython syntax for this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -VirtualHostMappingStep [
    [bundle_symbolic_name_1 bundle_version_1
    web_module_name_1 virtual_host_1]
    [bundle_symbolic_name_2 bundle_version_2
     web_module_name_2 virtual_host_2]]
  ...
]')
```

  For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/WAB.symbolic.name
```

  For example, for a composite bundle `com.ibm.ws.eba.helloWorldCBA` at version 1.0.0 that contains two WABs (`com.ibm.ws.eba.helloWorldService.web` at version 1.0.0, which is to be mapped to `default_host`, and `com.ibm.ws.eba.helloWorldService.withContextRoot` at version 0.9.0, which is to be mapped to `test_host`), this aspect of the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0 "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.withContextRoot
    0.9.0 "HelloWorld second service" test_host]]
  ...
]')
```

  Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Map virtual hosts for web modules.

- Add authentication aliases to Blueprint resource references in composite bundles.

  The Jython syntax for this aspect of the command is as follows.

```
AdminTask.addCompUnit('[
  ...
  -BlueprintResourceRefBindingStep [
    [
    bundle_symbolic_name
    bundle_version
    blueprint_resource_reference_id
    interface_name
    jndi_name
    authentication_type
    sharing_setting
    authentication_alias_name
    ]]
  ...
]')
```

Notes:

- For composite bundles, the bundle symbolic name has the following syntax:

  `CBA.symbolic.name_CBA.bundle.version/embedded_bundle.symbolic.name`
- The value for *jndi_name* must match the JNDI name that you declare in the **filter** attribute of the resource reference element in the Blueprint XML file.

For example, for a composite bundle `com.ibm.ws.eba.helloWorldCBA` at version 1.0.0 that contains a bundle `com.ibm.ws.eba.helloWorldService.properties.bundle.jar` at Version 1.0.0, which is to be bound to authentication alias `alias1`, the command is as follows:

```
AdminTask.addCompUnit('[
  ...
  -BlueprintResourceRefBindingStep[
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.properties.bundle
    1.0.0 resourceRef javax.sql.DataSource jdbc/Account Container Shareable alias1]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Add authentication aliases to Blueprint resource references.

- Bind web module message destination references to administered objects in composite bundles.

  The Jython syntax for this aspect of the command is as follows

```
AdminTask.addCompUnit('[
  ...
  -WebModuleMsgDestRefs [
    [
    bundle_symbolic_name
    bundle_version
    resource_reference_id
    resource_type
    target_jndi_name
    ]]
  ...
]')
```

  For composite bundles, the bundle symbolic name has the following syntax:

  `CBA.symbolic.name_CBA.bundle.version/WAB.symbolic.name`

  For example:

```
AdminTask.addCompUnit('[
  ...
  -WebModuleMsgDestRefs [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jms/myQ javax.jms.Queue
    jms/workQ]
```

```
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jms/myT javax.jms.Topic
    jms/notificationTopic]]
  ...
]')
```

Note: For bundles other than composite bundles, the syntax is slightly different. See Step: Bind web module message destination references to administered objects.

- Map web module resource references to resources in composite bundles.

  The Jython syntax for this aspect of the command is as follows.

```
AdminTask.addCompUnit('[
  ...
  -WebModuleResourceRefs [
    [
    bundle_symbolic_name
    bundle_version
    resource_reference_id
    resource_type
    target_jndi_name
    login_configuration
    login_properties
    extended_properties
    ]]
  ...
]')
```

  For composite bundles, the bundle symbolic name has the following syntax:

```
CBA.symbolic.name_CBA.bundle.version/WAB.symbolic.name
```

  For example:

```
AdminTask.addCompUnit('[
  ...
  -WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
  ...
]')
```

  Notes:

  – If you use multiple extended properties, the jython syntax is `"extprop1=extval1,extprop2=extval2"`.

  – For bundles other than composite bundles, the syntax is slightly different. See Step: Map web module resource references to resources.

Example

In the following example, the jython syntax from the previous steps is combined with the additional steps described in “Adding an EBA asset to a composition unit by using wsadmin commands” on page 2096 so that, by running the **addCompUnit** command once only, a composition unit is created and added to a business-level application. In the example, an EBA file `com.ibm.ws.eba.helloWorldService.eba` contains a composite bundle with symbolic name `com.ibm.ws.eba.helloWorldCBA`, at version `1.0.0`. This composite bundle contains two WABs:

- `com.ibm.ws.eba.helloWorldService.web`, at version 1.0.0

- com.ibm.ws.eba.helloWorldService.withContextRoot, at version 0.9.0

The composite bundle also contains a bundle with symbolic name com.ibm.ws.eba.helloWorldService.properties.bundle, at version 1.0.0.

```
AdminTask.addCompUnit('[
  -blaID WebSphere:blaname=helloWorldService
  -cuSourceID WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
  -CUOptions [
    [WebSphere:blaname=helloWorldService.eba
    WebSphere:assetname=com.ibm.ws.eba.helloWorldService.eba
    com.ibm.ws.eba.helloWorldService_0001.eba "" 1 false DEFAULT]]
  -MapTargets [[ebaDeploymentUnit WebSphere:cluster=cluster1]]
  -ActivationPlanOptions [[default ""]]
  -ContextRootStep [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web 1.0.0
    "/hello/web"]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.withContextRoot 0.9.0
    "/hello/service"]]
  -EJBMappingsStep [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0 ExchangeRate_ejb
      com.ibm.ws.eba.ejb.ExchangeRate Local ejb/ExchangeRate]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0 CustomerDetails_ejb
      com.ibm.ws.eba.ejb.CustomerDetails Remote ejb/CustomerDetails]]
  -EJBRefStep [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
      META-INF/ejb-jar.xml/CurrencyExchange ExchangeRate com.ibm.ws.eba.ejb.ExchangeRate
      ejb:ExchangeRate]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
      WEB-INF/web.xml CustomerDetails com.ibm.ws.eba.ejb.CustomerDetails ejb:CustomerDetails]]
  -EJBResourceRefs [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
     ExchangeRate dataSource1 javax.sql.DataSource ref/ds1 ClientContainer
     "WebSphere:name=mprop1,value=val1,description=desc1"
     "exprop1=expropval1+exprop2=expropval2"]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
     CustomerDetails dataSource2 javax.sql.DataSource ref/ds2 WSLogin "" ""]]
  -EJBMsgDestRefs [
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.currencyService 1.0.0
     ExchangeRate jms/myQ javax.jms.Queue jms/workQ]
    [com.ibm.ws.eba.financeCBA_1.0.0/com.ibm.ws.eba.accountService 0.9.0
     CustomerDetails jms/myT javax.jms.Topic jms/notificationTopic]]
  -VirtualHostMappingStep [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0 "HelloWorld service" default_host]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.withContextRoot
    0.9.0 "HelloWorld second service" test_host]]
  -MapRolesToUsersStep [
    [ROLE1 No Yes "" ""]
    [ROLE2 No No WABTestUser1 ""]
    [ROLE3 No No "" WABTestGroup1]
    [ROLE4 Yes No "" ""]]
  -MapRunAsRolesToUsersStep [
    [Role1 User1 password1]
    [AdminRole User3 password3]]
  -BlueprintResourceRefBindingStep[
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.properties.bundle
    1.0.0 resourceRef javax.sql.DataSource jdbc/Account Container Shareable alias1]]
  -WebModuleMsgDestRefs [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jms/myQ javax.jms.Queue
    jms/workQ]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
```

```
    1.0.0
    jms/myT javax.jms.Topic
    jms/notificationTopic]]
  -WebModuleResourceRefs [
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jdbc/jtaDs javax.sql.DataSource
    jdbc/helloDs "" "" ""]
    [com.ibm.ws.eba.helloWorldCBA_1.0.0/com.ibm.ws.eba.helloWorldService.web
    1.0.0
    jdbc/nonJtaDs javax.sql.DataSource
    jdbc/helloDsNonJta "" "" "extprop1=extval1"]]
]')
```

Debugging bundles at run time

You can explore or debug the bundles associated with a specific OSGi application or shared bundle framework by using either the WebSphere Application Server administrative console or the wsadmin-based OSGi Applications command-line console.

Debugging bundles at run time by using the WebSphere Application Server administrative console

The WebSphere Application Server administrative console provides panels that you can use to explore or debug a specific set of bundles running on an application server.

You can explore or debug bundles by interrogating the contents of OSGi *frameworks*. A framework contains a collection of bundles, together with the packages and services associated with the bundles. There are two types of framework:

Isolated framework
: An isolated framework contains the bundles that are defined exclusively for a specific application; each OSGi application runs in its own isolated framework. In a network deployment environment, there is one isolated framework for each server on which the application is installed. If an application includes one or more composite bundles, either as part of the application, or as an extension to the application, there is a separate isolated framework for each composite bundle.

Share bundle framework
: There is a shared bundle framework per server, containing all the shared bundles that are available to the applications that are installed on that server. If the applications indirectly reference one or more composite bundles through package dependencies, there is a shared bundle framework for each composite bundle.

You can find the state of the bundles in a framework, and see which bundles import or export certain packages, or which bundles register or consume a particular service. You can also see the values of the headers in the bundle manifest files.

You can navigate trails through bundles, packages, and services. For example, you might navigate the following trail:

1. Select a package.
2. See which bundle exports this package.
3. Find out the services that this bundle registers.
4. Find out which other bundles consume each of these services.
5. Explore the details of each of these bundles.

The panels have a breadcrumb trail that shows you where you are in a trail, and allows you to retrace the steps that you have taken through the trail.

Note: You can view the frameworks for an application only if the application has been started. If the application fails to start, the link to access the framework panels is not available.

The following sections describe all the panels, and the administrative console commands that you can use to display them.

Application OSGi frameworks

This panel lists all the OSGi frameworks that are associated with a given application.

To view this panel in the administrative console, click the following path:

Applications > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console**

For details of the information that is displayed in this panel, see Application OSGi frameworks [Collection].

Bundles in OSGi framework

This panel lists all the bundles in a given OSGi framework.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name***
- From the Packages in OSGi framework [Collection] page, or the Services in OSGi framework [Collection] page, click **Framework bundles**.

For details of the information that is displayed in this panel, see Bundles in OSGi framework [Collection].

Packages in OSGi framework

This panel lists all the packages in a given OSGi framework.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > **Framework packages**
- From the Bundles in OSGi framework [Collection] page, or the Services in OSGi framework [Collection] page, click **Framework packages**.

For details of the information that is displayed in this panel, see Packages in OSGi framework [Collection].

Services in OSGi framework

This panel lists all the services in a given OSGi framework.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > **Framework services**
- From the Packages in OSGi framework [Collection] page, or the Bundles in OSGi framework [Collection] page, click **Framework services**.

For details of the information that is displayed in this panel, see Services in OSGi framework [Collection].

Packages in bundle

This panel lists all the imported packages and exported packages for a given OSGi bundle.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > ***bundle_name*** > **Bundle packages**
- From the Packages in OSGi framework [Collection] page, or the Services in OSGi framework [Collection] page, click **Framework bundles** > ***bundle_name*** > **Bundle packages**.

For details of the information that is displayed in this panel, see Packages in bundle [Collection].

Services in bundle

This panel lists all the services that are registered, and all the services that are consumed, by a given OSGi bundle.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > ***bundle_name*** > **Bundle services**
- From the Packages in OSGi framework [Collection] page, or the Bundles in OSGi framework [Collection] page, click **Framework bundles** > ***bundle_name*** > **Bundle services**.

For details of the information that is displayed in this panel, see Services in bundle [Collection].

Bundle details

This panel displays the details of a given bundle, including identification information, the values of headers in the bundle manifest file, and bundle dependencies.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > ***bundle_name***
- From the Packages in OSGi framework [Collection] page, or the Services in OSGi framework [Collection] page, click **Framework bundles** > ***bundle_name***.

For details of the information that is displayed in this panel, see Bundle details [Settings].

Package details

This panel displays the details of a given package, including identification information, and the bundles that export and import the package.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > **Framework packages** > ***package_name***
- From the Bundles in OSGi framework [Collection] page, or the Services in OSGi framework [Collection] page, click **Framework packages** > ***package_name***.
- From the Bundle details [Settings] page, click **Bundle packages** > ***package_name***.

For details of the information that is displayed in this panel, see Package details [Settings].

Service details

This panel displays the details of a given service, including identification information, the service interfaces, the bundles that have registered or that use the service, and the service properties.

To view this panel in the administrative console, click one of the following paths:

- **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***composition_unit_name*** > **[Additional Properties] OSGi application console** > ***framework_name*** > **Framework services** > ***service_identifier***
- From the Packages in OSGi framework [Collection] page, or the Bundles in OSGi framework [Collection] page, click **Framework services** > ***service_identifier***.
- From the Bundle details [Settings] page, click **Bundle services** > ***service_identifier***.

For details of the information that is displayed in this panel, see Service details [Settings].

Debugging bundles at run time by using the command-line console

The OSGi Applications command-line console is a set of wsadmin commands that you can use to explore or debug a specific set of bundles running on an application server. As an aid to debugging applications in a test environment, the console also includes commands to start and stop bundles.

About this task

You can explore or debug bundles by interrogating the contents of OSGi *frameworks*. A framework contains a collection of bundles, together with the packages and services associated with the bundles. There are two types of framework:

Isolated framework
: An isolated framework contains the bundles that are defined exclusively for a specific application; each OSGi application runs in its own isolated framework. In a network deployment environment, there is one isolated framework for each server on which the application is installed. If an application includes one or more composite bundles, either as part of the application, or as an extension to the application, there is a separate isolated framework for each composite bundle.

Share bundle framework
: There is a shared bundle framework per server, containing all the shared bundles that are available to the applications that are installed on that server. If the applications indirectly reference one or more composite bundles through package dependencies, there is a shared bundle framework for each composite bundle.

To work with a framework that is currently running, you first connect to the framework.

Procedure

1. Start the OSGi Applications command-line console.

   At a command prompt, run the `osgiApplicationConsole.bat` command (for Windows systems) or the `osgiApplicationConsole.sh` command (for Linux systems). You can run this command from the *`app_server_root`*`/bin` directory or from any *`profile_root`*`/bin` directory. The command takes the following optional parameters:

   -h The host name of the target machine. For example, `machine1.hursley.ibm.com`.

   -o The port number of the SOAP port of the target server. For example, `8880`.

   -u The user ID, if the wsadmin connection is secured.

   -p The password, if the wsadmin connection is secured.

For example:

```
app_server_root/bin/osgiApplicationConsole -h machine1.hursley.ibm.com -o 8880
```

The wsadmin command prompt is displayed. This instance of wsadmin recognizes OSGi Applications console commands.

2. Connect to an available framework.

   You use the **connect** command to connect to a specific framework.

   If you know the framework name and version number, and the node and server on which the framework is running, you can use this information to connect. For example:

   ```
   wsadmin>connect("com.ibm.ws.eba.helloWorldService.eba", "1.0.0", "wasNode1", "server1")
   ```

   Alternatively, use the **list** command to list all available frameworks and to provide a unique ID for each framework, then use this ID to connect. For example:

   a. List all available frameworks:

      ```
      wsadmin>list()
      ```

      If you are connecting to an individual application server, this command might generate the following system response:

      ```
      ID   Bundle                                Version  Node      Server
      0    SharedBundles                         7.0.0    wasNode1  server1
      1    com.ibm.ws.eba.helloWorldService.eba  1.0.0    wasNode1  server1
      2    com.ibm.ws.eba.obr.fep.eba5.eba       1.0.0    wasNode1  server1
      3    com.ibm.ws.eba.wab.componenttest      1.0.0    wasNode1  server1
      ```

   b. Connect to the com.ibm.ws.eba.helloWorldService.eba framework:

      ```
      wsadmin>connect(1)
      ```

      If the command completes successfully it generates the following system responses:

      ```
      CWSAJ0035I: Connecting to framework com.ibm.ws.eba.helloWorldService.eba_1.0.0
      on node wasNode1 and server server1.
      CWSAJ0036I: Successfully connected to framework com.ibm.ws.eba.helloWorldService.eba_1.0.0.
      ```

   To connect to a different framework, run the **connect** command again. You do not need to disconnect from one framework before connecting to another framework.

3. Work with the connected framework.

   Use one or more of the following commands to work with the framework to which you are connected:

   - List all available frameworks, and indicate the currently connected framework.

     Use the **list** command:

     ```
     wsadmin>list()
     ```

     This command might generate the following system response:

     ```
     ID   Bundle                                Version  Node      Server
     0    SharedBundles                         7.0.0    wasNode1  server1
     1    com.ibm.ws.eba.helloWorldService.eba  1.0.0    wasNode1  server1  <== Connected
     2    com.ibm.ws.eba.obr.fep.eba5.eba       1.0.0    wasNode1  server1
     3    com.ibm.ws.eba.wab.componenttest      1.0.0    wasNode1  server1
     ```

   - Display summary information about each of the bundles in the framework.

     Use the **ss** command:

     ```
     wsadmin>ss()
     ```

     Note: “ss” stands for “short status”.
     This command might generate the following system response:

     ```
     ID      State       Bundle
     0       ACTIVE      org.eclipse.osgi_3.6.1.R36x_v20100806
     1       ACTIVE      com.ibm.samples.websphere.osgi.blog.app_1.0.0
     2       ACTIVE      com.ibm.samples.websphere.osgi.blog_1.0.0
     ```

```
3       ACTIVE      com.ibm.samples.websphere.osgi.blog.persistence_1.0.0
4       ACTIVE      com.ibm.samples.websphere.osgi.blog.web_1.0.0
5       ACTIVE      com.ibm.samples.websphere.osgi.blog.api_1.0.0
```

- Display comprehensive information about all of the bundles in the framework, and the services that they register or consume.

  Use the **bundles** command:

  ```
  wsadmin>bundles()
  ```

  This command might generate the following system response.

  Note: Information about the bundles that you included in your application is listed as bundle 2 and subsequent bundles. Bundles 0 and 1 are system artifacts; do not modify or rely on these bundles. Although the detail for bundle 0 is not shown in the following example, this bundle typically generates much more information than the other bundles.

```
org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  Id=0, Status=ACTIVE  Location=System Bundle
...
com.ibm.samples.websphere.osgi.blog.app_1.0.0 [1]
  Id=1, Status=ACTIVE  Location=com.ibm.samples.websphere.osgi.blog.app_1.0.0
...
com.ibm.samples.websphere.osgi.blog_1.0.0 [2]
  Id=2, Status=ACTIVE
Location=reference:file:/C:/IBM/WebSphere/AppServer/profiles/profile01/
installedEBAs/com.ibm.samples.websphere.osgi.blog.app_1.0.0/byValue/
98b31e7a-4375-45fa-be20-d34b06f5c8b8.3/3/
  Registered Services

{com.ibm.samples.websphere.osgi.blog.api.BloggingService}={service.id=46,
osgi.service.blueprint.compname=bloggingServiceComponent
}

{org.osgi.service.blueprint.container.BlueprintContainer}={osgi.blueprint.
container.symbolicname=com.ibm.samples.websphere.osgi.blog,service.id=47,
osgi.blueprint.container.version=1.0.0}
  No services in use.
com.ibm.samples.websphere.osgi.blog.persistence_1.0.0 [3]
  Id=3, Status=ACTIVE
Location=reference:file:/C:/IBM/WebSphere/AppServer/profiles/profile01/
installedEBAs/com.ibm.samples.websphere.osgi.blog.app_1.0.0/byValue/
98b31e7a-4375-45fa-be20-d34b06f5c8b8.1/1/
  Registered Services

{javax.persistence.EntityManagerFactory}={osgi.unit.provider=org.apache.openjpa.
persistence.PersistenceProviderImpl,service.id=42,osgi.unit.name=blogExample,
osgi.unit.version=1.3.0,org.apache.aries.jpa.container.managed=true,org.apache.
aries.jpa.default.unit.name=false}

{javax.persistence.EntityManagerFactory}={osgi.unit.provider=org.apache.openjpa.
persistence.PersistenceProviderImpl,service.id=43,org.apache.aries.jpa.proxy.factory=true,
osgi.unit.name=blogExample,osgi.unit.version=1.3.0,org.apache.aries.jpa.container.managed=true,
org.apache.aries.jpa.default.unit.name=false}

{com.ibm.samples.websphere.osgi.blog.persistence.api.BlogPersistenceService}={service.id=44,
osgi.service.blueprint.compname=persistenceImpl}

{org.osgi.service.blueprint.container.BlueprintContainer}={osgi.blueprint.container.
symbolicname=com.ibm.samples.websphere.osgi.blog.persistence,service.id=45,
osgi.blueprint.container.version=1.0.0}
  No services in use.
com.ibm.samples.websphere.osgi.blog.web_1.0.0 [4]
  Id=4, Status=ACTIVE
Location=reference:file:/C:/IBM/WebSphere/AppServer/profiles/profile01/
installedEBAs/com.ibm.samples.websphere.osgi.blog.app_1.0.0/byValue/
98b31e7a-4375-45fa-be20-d34b06f5c8b8.2/2/
```

```
  Registered Services

{javax.servlet.ServletContext}={service.id=48,osgi.web.contextpath=/blog,
osgi.web.version=1.0.0,osgi.web.symbolicname=com.ibm.samples.websphere.osgi.blog.web}

{org.osgi.service.blueprint.container.BlueprintContainer}={osgi.blueprint.container.
symbolicname=com.ibm.samples.websphere.osgi.blog.web,service.id=49,
osgi.blueprint.container.version=1.0.0}
  No services in use.
com.ibm.samples.websphere.osgi.blog.api_1.0.0 [5]
  Id=5, Status=ACTIVE
Location=reference:file:/C:/IBM/WebSphere/AppServer/profiles/profile01/
installedEBAs/com.ibm.samples.websphere.osgi.blog.app_1.0.0/byValue/
98b31e7a-4375-45fa-be20-d34b06f5c8b8.0/0/
  No registered services.
  No services in use.
```

- Display comprehensive information about a given bundle in the framework.

  Use the **bundle** command. Specify the ID of the bundle that you want to examine. The bundle ID values are one of the outputs of the **ss** command.

  This command lists the information associated with the specified bundle. For example the bundle symbolic name, the bundle version, the services that the bundle registers, the services that are consumed by the bundle, and whether the bundle is a fragment or a host bundle.

  For example:

  ```
  wsadmin>bundle(3)
  ```

  This command might generate the following system response:

  ```
  com.ibm.samples.websphere.osgi.blog.persistence_1.0.0 [3]
    Id=3, Status=ACTIVE
  Location=reference:file:/C:/IBM/WebSphere/AppServer/profiles/profile01/
  installedEBAs/com.ibm.samples.websphere.osgi.blog.app_1.0.0/byValue/
  98b31e7a-4375-45fa-be20-d34b06f5c8b8.1/1/
    Registered Services

  {javax.persistence.EntityManagerFactory}={osgi.unit.provider=org.apache.openjpa.
  persistence.PersistenceProviderImpl,service.id=42,osgi.unit.name=blogExample,
  osgi.unit.version=1.3.0,org.apache.aries.jpa.container.managed=true,org.apache.
  aries.jpa.default.unit.name=false}

  {javax.persistence.EntityManagerFactory}={osgi.unit.provider=org.apache.openjpa.
  persistence.PersistenceProviderImpl,service.id=43,org.apache.aries.jpa.proxy.factory=true,
  osgi.unit.name=blogExample,osgi.unit.version=1.3.0,org.apache.aries.jpa.container.managed=true,
  org.apache.aries.jpa.default.unit.name=false}

  {com.ibm.samples.websphere.osgi.blog.persistence.api.BlogPersistenceService}={service.id=44,
  osgi.service.blueprint.compname=persistenceImpl}

  {org.osgi.service.blueprint.container.BlueprintContainer}={osgi.blueprint.container.
  symbolicname=com.ibm.samples.websphere.osgi.blog.persistence,service.id=45,
  osgi.blueprint.container.version=1.0.0}
    No services in use.
    No exported packages
    Imported Packages
    Imported Packages
      com.ibm.samples.websphere.osgi.blog.persistence.api;
      version="1.0.0"<com.ibm.samples.websphere.osgi.blog.api_1.0.0 [3]>
      javax.persistence; version="1.1.0"<org.eclipse.osgi_3.6.1.R36x_v20100806 [3]>
    No fragment bundles
    No host bundles
    No named class spaces
    Required bundles
      org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
      com.ibm.samples.websphere.osgi.blog.api_1.0.0 [5]
  ```

- Display header information about a given bundle.

Use the **headers** command. Specify the ID of the bundle that you want to examine. The bundle ID values are one of the outputs of the **ss** command.

For example:

```
wsadmin>headers(2)
```

This command might generate the following system response:

```
Ant-Version = Apache Ant 1.7.1
Bundle-ActivationPolicy = lazy
Bundle-ManifestVersion = 2
Bundle-Name = Blog Core Services Bundle
Bundle-SymbolicName = com.ibm.samples.websphere.osgi.blog
Bundle-Vendor = IBM
Bundle-Version = 1.0.0
Created-By = 2.4 (Your Corporation)
Import-Package = com.ibm.samples.websphere.osgi.logging;version="[1.0.0,1.1.0)",
com.ibm.samples.websphere.osgi.blog.api;version="[1.0.0,1.1.0)",
com.ibm.samples.websphere.osgi.blog.comment.persistence.api;version="[1.0.0,1.1.0)",
com.ibm.samples.websphere.osgi.blog.persistence.api;version="[1.0.0,1.1.0)"
Manifest-Version = 1.0
```

- Display information about the packages that are imported or exported by the framework.

  Use the **packages** command. Optionally, specify either or both of the following parameters to select a particular package or subset of packages:

  bundle ID
  Display information about the exported packages for this bundle.

  The bundle ID values are one of the outputs of the **ss** command.

  package name
  Display information about the specified package.

  Command syntax:

  ```
  wsadmin>packages()
  wsadmin>packages(bundle_id)
  wsadmin>packages(package_name)
  ```

  Example using a bundle ID:

  ```
  wsadmin>packages(5)
  ```

  This command might generate the following system response:

  ```
  com.ibm.samples.websphere.osgi.blog.persistence.api; version="1.0.0"
  <com.ibm.samples.websphere.osgi.blog.api_1.0.0 [5]>
    com.ibm.samples.websphere.osgi.blog_1.0.0 [2] imports
    com.ibm.samples.websphere.osgi.blog.persistence_1.0.0 [3] imports
  com.ibm.samples.websphere.osgi.blog.comment.persistence.api; version="1.0.0"
  <com.ibm.samples.websphere.osgi.blog.api_1.0.0 [5]>
    com.ibm.samples.websphere.osgi.blog_1.0.0 [2] imports
  com.ibm.samples.websphere.osgi.blog.api; version="1.0.0"
  <com.ibm.samples.websphere.osgi.blog.api_1.0.0 [5]>
    com.ibm.samples.websphere.osgi.blog_1.0.0 [2] imports
    com.ibm.samples.websphere.osgi.blog.web_1.0.0 [4] imports
  ```

  Example using a package name:

  ```
  wsadmin>packages("com.ibm.samples.websphere.osgi.blog.comment.persistence.api")
  ```

  This command might generate the following system response:

  ```
  com.ibm.samples.websphere.osgi.blog.comment.persistence.api; version="1.0.0"
  <com.ibm.samples.websphere.osgi.blog.api_1.0.0 [5]>
    com.ibm.samples.websphere.osgi.blog_1.0.0 [2] imports
  ```

- Display information about the services that are currently registered by the framework. For each service this information includes the service interfaces, the bundle that registered the service, and any bundles that consume the service.

Use the **services** command. Optionally, specify either or both of the following parameters to select a particular service or subset of services:

service ID
Display information about a specified service.

filter
Display information about all services that match the filter.

The filter must comply with the OSGi filter format as defined in section 3.2.7 of the OSGi Service Platform Release 4 Version 4.2 Core Specification.

Command syntax:

```
wsadmin>services()
wsadmin>services(service_id)
wsadmin>services("(&(prop1=value_1)(prop2=value_2))")
```

Example using a service ID:

```
wsadmin>services(2)
```

This command might generate the following system response:

```
{org.osgi.service.packageadmin.PackageAdmin}={service.id=2,
service.ranking=2147483647,service.vendor=Eclipse.org - Equinox,
service.pid=0.org.eclipse.osgi.framework.internal.core.PackageAdminImpl}
  Registered by bundle: org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  Bundles using service:
    org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
```

Example using a single property filter:

```
wsadmin>services("(objectClass=org.osgi.service.packageadmin.PackageAdmin)")
```

This command might generate the following system response:

```
{org.osgi.service.packageadmin.PackageAdmin}={service.id=2,
service.ranking=2147483647,service.vendor=Eclipse.org - Equinox,
service.pid=0.org.eclipse.osgi.framework.internal.core.PackageAdminImpl}
  Registered by bundle: org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  Bundles using service:
    org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
```

Example using a multiple property filter:

```
wsadmin>services("(&(objectClass=javax.resource.Referenceable)(ibm.private.jndi.object=true))")
```

This command might generate the following system response:

```
{java.lang.reflect.InvocationHandler,com.ibm.websphere.rsadapter.WSDataSource,java.sql.Wrapper,
javax.sql.CommonDataSource,javax.resource.Referenceable,javax.sql.DataSource}={service.id=35,
osgi.jndi.service.name=jdbc/pgc,ibm.private.jndi.object=true}
  Registered by bundle: org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  No bundles using service.
{java.lang.reflect.InvocationHandler,com.ibm.websphere.rsadapter.WSDataSource,java.sql.Wrapper,
javax.sql.CommonDataSource,javax.resource.Referenceable,javax.sql.DataSource}={service.id=36,
osgi.jndi.service.name=jdbc/DefaultEJBTimerDataSource,ibm.private.jndi.object=true}
  Registered by bundle: org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  No bundles using service.
{java.lang.reflect.InvocationHandler,com.ibm.websphere.rsadapter.WSDataSource,java.sql.Wrapper,
javax.sql.CommonDataSource,javax.resource.Referenceable,javax.sql.DataSource}={service.id=39,
osgi.jndi.service.name=jdbc/lrsched,ibm.private.jndi.object=true}
  Registered by bundle: org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  No bundles using service.
{javax.resource.Referenceable,javax.resource.cci.ConnectionFactory}={service.id=40,
osgi.jndi.service.name=eis/jdbc/pgc_CMP,ibm.private.jndi.object=true}
  Registered by bundle: org.eclipse.osgi_3.6.1.R36x_v20100806 [0]
  No bundles using service.
```

- Update the cached runtime information about the framework.

It can take some time for the system to gather the runtime information about the framework and the bundles running within it. Therefore, when you run the `osgiApplicationConsole` command, the command-line console caches the runtime information. If you make changes to the currently selected framework, for example by updating or adding bundles, these changes are not reflected in the output of the command-line console commands until you run the **refresh** command.

Note: You can also update this runtime information by restarting the OSGi Applications command-line console. However, you then have to reconnect to the framework. It is quicker to use the **refresh** command.

Command syntax:

```
wsadmin>refresh()
```

- Stop a bundle.

  Use the **stop** command to stop the bundle specified by a given bundle ID. The bundle ID values are one of the outputs of the **ss** command.

  Note:

  – The OSGi Applications command-line console is intended primarily as a method of debugging applications. In a production environment, you would normally use the administrative console or equivalent administrative commands to start and stop applications.

  – Do not use the **stop** command on any bundles under the shared bundle framework, on system bundles, or on the bundles that represent system artifacts (bundle ID 0 and 1), otherwise serious errors can result.

  For example:

  ```
  wsadmin>stop(5)
  ```

  This command might generate the following system response:

  ```
  CWSAJ0042I: Stopping Bundle com.ibm.samples.websphere.osgi.blog.api_1.0.0.
  CWSAJ0034I: Bundle com.ibm.samples.websphere.osgi.blog.api stopped successfully.
  ```

- Start a bundle.

  Use the **start** command to start the bundle specified by a given bundle ID. The bundle ID values are one of the outputs of the **ss** command.

  Note:

  – The OSGi Applications command-line console is intended primarily as a method of debugging applications. In a production environment, you would normally use the administrative console or equivalent administrative commands to start and stop applications.

  – Do not use the **start** command on any bundles under the shared bundle framework, on system bundles, or on bundles that represent applications (bundle ID 0 and 1), otherwise serious errors can result.

  For example:

  ```
  wsadmin>start(5)
  ```

  This command might generate the following system response:

  ```
  CWSAJ0040I: Starting Bundle com.ibm.samples.websphere.osgi.blog.api_1.0.0.
  CWSAJ0032I: Bundle com.ibm.samples.websphere.osgi.blog.api started successfully.
  ```

- Display help information about the console commands.

  Use the **help** command to display a summary of the runtime commands and their uses.

  Command syntax:

  ```
  wsadmin>help()
  ```

This command generates the following system response:

```
CWSAJ0025I:      OSGi application console

Display commands: These commands work only if connected to a framework

ss()                - This command gives the summary information about the installed bundles.
bundles()           - This command gives comprehensive information about the installed bundles.
packages()          - This command gives information about the imported/exported packages.
services()          - This command gives information about the registered Services.

bundle(<bundleID>)  - This command gives information about the specified bundle
headers(<bundleID>) - This command gives information about the headers associated with the
                      specified bundle
packages(<bundle ID>) - This command gives information about the exported packages for this bundle.
packages(<package Name>) - This command gives information about the specified package.
services(<service ID>) - This command gives information about the specified service.
services(<OSGI Filter>) - This command gives information about the services matching the filter.

refresh() - This command refreshes the internal OSGi application console cache with the latest
            information about the state of the framework.

Framework commands:

list() - This command lists the available frameworks that you can connect to.
connect(<Framework id>) - This command connects to the specified framework.
connect(<Bundle Name>, <Bundle Version>, <Node Name>, <Server Name>)
                 - This command connects to the specified framework.

Controlling Bundles:

start(<bundleID>) - This command starts the requested bundle.
stop(<bundleID>   - This command stops the requested bundle.
```

Chapter 45. Deploying SCA composites

This page provides a starting point for finding information about Service Component Architecture (SCA) composites, which consist of components that implement business functions in the form of services.

You typically do not deploy SCA composites directly onto a product server. To deploy SCA composites, you import SCA composites as assets to the product repository and add the assets to business-level applications.

Deploying SCA business-level applications

Deploying an Service Component Architecture (SCA) business-level application consists of creating an empty business-level application and then adding SCA assets, shared libraries, business-level applications, and other artifacts as composition units to the empty business-level application.

Before you begin

Develop the artifacts to go in the application and configure the target server or cluster. You must deploy SCA composite assets of a business-level application to a Version 8.x target or to a Version 7.0 target that is enabled for the Feature Pack for SCA.

If your SCA composite or application uses OASIS support, you must deploy the SCA asset or application to a Version 8.5 target.

If your SCA composite or application uses Feature Pack for SCA Version 1.0.1 functionality, you must deploy the SCA asset or application to a Version 8.x target or to a feature pack Version 1.0.1.0 target. Version 1.0.1 functionality includes:

- Java Message Service (JMS) bindings
- Atom bindings
- HTTP bindings with a wire format of JSON-RPC
- Java Platform, Enterprise Edition (Java EE) integration modules (`implementation.jee`, `implementation.web`, or `implementation.ejb` components)
- SCA Spring component implementations
- OSGi applications as SCA component implementations
- Service Data Objects (SDO) composites

About this task

When creating a business-level application, you can configure the application enough to enable it to run on the server. Later, you can configure the application and its contents further, start or stop the application, and otherwise manage its activity.

The topics in this section describe how to deploy and administer a business-level application or its contents using the administrative console. You can also use programming or wsadmin scripting.

Procedure

- Import assets to a repository.
- Create an SCA business-level application that has SCA assets, shared libraries, or business-level applications.
- Start the application.
- Stop the application.
- Update SCA composite artifacts.
- Update the application and its configuration units.

- View the composite definition of an SCA asset composition unit.
- View SCA domain information.
- View or edit JMS bindings on references and services of SCA composites.
- Delete the application.

What to do next

After making changes to administrative configurations of your applications in the administrative console, ensure that you save the changes.

Importing assets

You must register application business logic such as Java Platform, Enterprise Edition (Java EE) archives, libraries, and other resource files with the product configuration as assets before you can add the assets to one or more business-level applications. Importing an asset registers it with the product configuration.

Before you begin

This topic assumes that you have one or more application binary files that you want to add to a business-level application. You must register those binary files as assets before you can add them to the business-level application.

About this task

Before a business-level application that uses an asset can be started on the target run time, the asset binaries must be extracted to a deployer-defined location on the file system that is local to the target run time. Importing an asset extracts binaries to a location that is local to the target run time.

The application server run time that reads the asset binaries either at application start time or while serving an incoming client request determines the extraction format of the asset binaries. The extraction format might include unzipping of Java archive (JAR) or compressed (zip) files.

This topic describes how to import an asset using the administrative console. Alternatively, you can use the wsadmin tool or programming.

Procedure

1. Click **Applications** > **New Application** > **New Asset** in the console navigation tree.
2. On the Upload asset page, specify the asset package to import.
   a. Specify the full path name of the asset.
   b. Click **Next**.
3. On the Select options for importing an asset page, specify asset settings.
   You typically can click **Next** and use the default values.
   a. Optional: For **Asset description**, specify a brief description of the asset.
   b. Optional: For **Asset binaries destination URL**, specify the target location of the asset.
      This setting specifies the location to which the product extracts the asset. After an asset is imported, the product looks for the asset in this location when a running application uses the asset.
      If you do not specify a value, the product installs the asset to the default location, `${`*profile_root*`}/installedAssets/`*asset_name*`/BASE/`.
   c. Optional: For **Asset type aspects**, examine the asset content type and version specified by the product. You cannot change this setting value.
      The type aspect typically denotes the type of application contents, such as a specification to which the application is written. For example, an enterprise bean (EJB) that supports the EJB Version 2.0 specification has the aspects `type=EJB,version=2.0`.

If the type aspect is `none` and if the asset is a JAR file, then the product associates a `javarchive` type aspect with the asset by default.

d. For **File permissions**, specify any file permissions that are set on asset binary files so the target run time can read or run the asset. Importing the asset extracts its binary files on the disk local to the target runtime environment.

   Try importing the asset using the default value. For detailed information on the **File permissions** setting, refer to the Select options for importing an asset page online help.

   Restriction: OSGi applications do not use a **File permissions** setting.

e. For **Current asset relationships**, add assets that the asset you are importing needs to run or remove assets that are not needed.

   When the product imports a JAR asset, the product detects asset relationships automatically by matching the dependencies defined in the JAR manifest with the assets that are already imported into the administrative domain.

f. For **Validate asset**, specify whether the product validates the asset.

   The setting is deselected by default. This **false** (**no**) value is appropriate for most assets. Only select **true** (**yes**) to validate an asset when needed.

   The product does not save the value specified for **Validate asset**. Thus, if you select to validate the asset (**yes**) now and later update the asset, when you update the asset you must enable this setting again for the product to validate the updated files.

   Restriction: OSGi applications do not use a **Validate asset** setting.

g. Click **Next**.

4. On the Summary page, click **Finish**.

Results

Several messages are displayed, indicating whether your asset is imported successfully.

An asset can contain multiple deployable objects as defined by the application contents of that asset. A *deployable object* is a part of the asset that you can map to a deployment target such as an application server or a cluster. If the product imports the asset successfully, then appropriate deployable objects are identified in the asset and are further used when a composition unit is created from that asset.

If the asset importing is not successful, read the messages and try importing the asset again. Correct the values noted in the messages.

What to do next

If the product imports the asset successfully and displays the list of assets on the Assets page, then click **Save**.

Add a composition unit to a business-level application using the asset that you imported. An asset included in a business-level application is represented by a composition unit.

Upload asset settings

Use this page to specify the asset to register with the asset repository. You can add registered assets to a business-level application.

To view this administrative console page, click **Applications** > **New application** > **New Asset**.

Importing an asset registers the asset with the asset repository.

The product manages the contents of a registered asset as a single entity. The contents of a registered asset must be accessible to application servers, web servers and other runtime environments that use the asset.

During asset importing, asset files typically are uploaded from a client workstation running the browser to the server running the administrative console, where they are registered. In such cases, use the web browser running the administrative console to select files to upload to the server.

Path to the asset:

Specifies the fully qualified path to the asset.

Specify one of the following supported assets:
- A single file, such as an enterprise bean (EJB) file
- An archive of files, such as a Java archive (JAR) or a compressed `.zip` file
- An archive of archives, such as an enterprise archive (EAR) or shared library file

Use **Local file system** if the browser and asset files are on the same machine (whether or not the server is on that machine, too).

Use **Remote file system** if the asset file resides on any node in the current cell context. Only supported assets are shown during the browsing. Also use **Remote file system** to specify an asset file that is already residing on the machine running the application server. For example, the field value might be *profile_root*`/installableApps/my_bean.ejb`. After the asset file is transferred, the **Remote file system** value shows the path of the temporary location on the server.

Asset settings

Use this page to specify options for the registration of an asset with the asset repository. Default values for the options are used if you do not specify a value. If the asset is an OSGi application, additional information about bundle download status is displayed.

To view this administrative console page, click **Applications** > **Application Types** > **Assets** > ***asset_name***. This page is similar to the Select options for importing an asset page on the asset import and update wizards.

Asset name:

Specifies a logical name for the asset. An asset name must be unique within a cell and cannot contain an unsupported character.

An asset name cannot begin with a period (`.`), cannot contain leading or trailing spaces, and cannot contain any of the following characters:

Table 272. Characters that you cannot use in a name. The product does not support these characters in a name.

Unsupported characters		
/ forward slash	**$** dollar sign	**'** single quote mark
**** backslash	**=** equal sign	**"** double quote mark
***** asterisk	**%** percent sign	**|** vertical bar
, comma	**+** plus sign	**<** left angle bracket
: colon	**@** at sign	**>** right angle bracket
; semi-colon	**#** hash mark	**&** ampersand (and sign)
? question mark	**]]>** No specific name exists for this character combination	

This **Asset name** field is the same as the **Name** setting on an Assets page.

Information	Value
Data type	String

Asset description:

Specifies a description for the asset.

Asset binaries destination URL:

Specifies the directory to which the product imports the asset file.

Information	Value
Data type	String
Units	Full path name

Asset type aspects:

Specifies the type of asset content. Examples of asset type include Java archive (JAR) files, shared libraries, enterprise application archive (EAR) files, and enterprise bundle archive (EBA) files.

The asset type suggests the content of the asset. An asset packaged as a JAR file might contain a web module, portlet, or web service. An asset packaged as an EBA file contains an OSGi application.

This setting is read-only. You cannot edit this setting.

Information	Value
Data type	String
Units	File type
Default	none

File permissions:

Specifies access permissions for asset binaries that the product expands to the asset binaries destination URL.

Restriction: OSGi applications do not use a **File permissions** setting.

You can specify file permissions in the text field. You can also set some of the commonly used file permissions by selecting them from the list. List selections overwrite file permissions set in the text field.

You can set one or more of the following file permission strings in the list. Selecting multiple options combines the file permission strings.

Table 273. File permission string sets for list options. Select a list option or specify a file permission string in the text field.

Multiple-selection list option	File permission string set
Allow all files to be read but not written to	`.*=755`
Allow executables to execute	`.*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755`
Allow HTML and image files to be read by everyone	`.*\.htm=755#.*\.html=755#.*\.gif=755#.*\.jpg=755`

Instead of using the multiple-selection list to specify file permissions, you can specify a file permission string in the text field. File permissions use a string that has the following format:

file_name_pattern=permission#file_name_pattern=permission

where *file_name_pattern* is a regular expression file name filter (for example, `.*\\.jsp` for all JSP files), *permission* provides the file access control lists (ACLs), and `#` is the separator between multiple entries of *file_name_pattern* and *permission*. If `#` is a character in a *file_name_pattern* string, use `\#` instead.

If multiple file name patterns and file permissions in the string match a uniform resource identifier (URI) within the asset, then the product uses the most stringent applicable file permission for the file. For example, if the file permission string is `.*\\.jsp=775#a.*\\.jsp=754`, then the `abc.jsp` file has file permission 754.

Tip: Using regular expressions for file matching pattern compares an entire string URI against the specified file permission pattern. You must provide more precise matching patterns using regular expressions as defined by Java programming API. For example, suppose the product processes the following directory and file URIs during a file permission operation:

Table 274. Example URIs for file permission operations. Results are shown following this table.

Number	Example URL
1	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war
2	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
3	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF/MANIFEST.MF
4	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/WEB-INF/classes/MyClass.class
5	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/mydir/MyClass2.class
6	/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/META-INF

The file pattern matching results are:
- `MyWarModule.war` does not match any of the URIs
- `.*MyWarModule.war.*` matches all URIs
- `.*MyWarModule.war$` matches only URI 1
- `.*\\.jsp=755` matches only URI 2
- `.*META-INF.*` matches URIs 3 and 6
- `.*MyWarModule.war/.*/.*\.class` matches URIs 4 and 5

If you specify a directory name pattern for **File permissions**, then the directory permission is set based on the value specified. Otherwise, the **File permissions** value set on the directory is the same as its parent. For example, suppose you have the following file and directory structure:

```
/opt/WebSphere/profiles/AppSrv01/installedApps/MyCell/MyApp.ear/MyWarModule.war/MyJsp.jsp
```

and you specify the following file pattern string:

```
.*MyApp.ear$=755#.*\.jsp=644
```

The file pattern matching results are:
- Directory `MyApp.ear` is set to 755
- Directory `MyWarModule.war` is set to 755
- Directory `MyWarModule.war` is set to 755

Important: Regardless of the operation system, always use a forward slash (/) as a file path separator in file patterns.

Access permissions specified here are at the asset level. You can also specify access permissions for asset binaries in the node-level configuration. The node-level file permissions specify the maximum (most lenient) permissions that can be given to asset binaries. Access permissions specified here at asset level can only be the same as or more restrictive than those specified at the node level.

Information	Value
Data type	String

Current asset relationships:

Specifies the assets to which this asset is related.

To add or remove a relationship, use the Manage relationships page:

1. Click **Manage Relationships**. The **Selected** list on the right lists the current asset relationships.
2. To add a relationship, select an asset in the **Available** list on the left and click **>>**.
3. To remove a relationship, select an asset in the **Selected** list on the right and click **<<**.
4. Click **OK**.

Information	Value
Data type	String
Default	none

Validate asset:

Specifies whether the product examines the asset references specified during asset importing or updating and, if validation is enabled, warns you of incorrect references or fails the operation.

Restriction: OSGi applications do not use a **Validate asset** setting.

An asset typically refers to resources using data sources for container-managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the asset is defined in the scope of the deployment target of that asset.

Select `true` (enable the check box) for resource validation and to stop operations that fail as a result of incorrect resource references. Select `false` (empty check box) for no resource validation.

Information	Value
Data type	String
Default	false (empty check box)

EBA Dependencies:

For an enterprise bundle archive (EBA) asset, displays the current bundle download status for all bundles in the asset. This item is only displayed if your asset is an EBA asset, which means that it contains an OSGi application.

You cannot update an EBA asset until bundle downloads are complete from any previous update, and until the business-level application that uses the asset has picked up the previous updates by being restarted. Before you try and update bundle versions, you can use the EBA dependency information to check the bundle download status of the asset. The status displayed is one of the following values:

- Bundles downloading...
- Bundle downloads are complete.
- No bundles downloads are required.

Note: In addition to the information given here, you can also check the bundle download status indirectly, by checking the status of the associated EBA composition unit as described in Checking and updating the EBA asset version used by a business-level application.

If bundle downloads for the asset are complete, or no bundle downloads are required, you can update the asset using either of the methods described in Maintaining bundle versions for an EBA asset.

If bundle downloads for the asset are complete, and a new version of the EBA asset is available, restart the business-level application to bring the EBA composition unit up-to-date and to run the newer configuration.

SCA application package deployment

The product supports deployment of many types of Service Component Architecture (SCA) artifacts as composition units of business-level applications. Typical artifacts include Java archive (JAR) files, compressed `.zip` files, and web application archive (WAR) files.

The following outlines the details about deployment of SCA artifacts:
- Deployment of JAR or compressed files
- Deployment of WAR files
- Notes and limitations

Deployment of JAR or compressed files

- The product supports one composite file for each package. The product determines which composite file to support using the following process:
  1. The SCA deployment extension looks for the `META-INF/sca-contribution.xml` file, gets the name of each deployable composite defined in the file, and uses QName values to find the actual composite file names under any directory for that composite. If more than one composite is found in the `sca-contribution.xml` file, you can select the composite to deploy.
  2. If there is no `META-INF/sca-contribution.xml` file defined, the SCA deployment extension looks for a composite file in the `META-INF/sca-deployables` directory.
- The product validates SCA composites for inconsistencies with SCA specifications.

  One specification requirement is that the names of top-level components must be unique. Thus, the product validates top-level component name uniqueness.

  Tip: Top-level components are also called domain components, with the top-level or domain typically the cell on multiple-server installations and the server scope on single-server installations.

  The product validates all composite files in a JAR or compressed file, regardless of the file location in the archive or whether the `sca-contribution.xml` file references the composite file. The product does not validate all services and references.

  The product writes warning and error messages resulting from the validation tests to the `SystemOut.log` file. Refer to the log file to learn about inconsistencies with specifications in your SCA composites.

  Note: This topic references one or more of the application server log files. As a recommended alternative, you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files on distributed and IBM i systems. You can also use HPEL in conjunction with your native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.
- The product uses the following process for QName resolution:

  - The product uses the QName to resolve composite files included in the top-level composite that use the element. For example, the `<include name="mynamespace:MyService"/>` statement looks for a composite file whose composite name is `MyService` and whose targetNameSpace is `mynamespace`. The following rules apply:
    - **name**: Use the outer composite.
    - **namespace declarations**: Merged into the outer composite.
    - **targetNamespace**: Use the outer composite (must be the same).
    - **local**: Use the composite (preferably the same but not required).
    - **requires(intents) and policySets**: Must be compatible, and aggregated into the outer composite.

    Deployable composite files must have name and targetNamespace values. The name and targetNamespace values are combined to form the QName of a composite file.
  - When a composite is used as a component implementation in the top-level composite, the composite is also resolved using the QName. For example, the `<implementation.composite name="mynamespace:MyComposite"/>` statement causes the product administration to look for a composite file whose composite name is `MyComposite` and whose targetNamespace is `mynamespace`.
- A JAR file can contain other JAR files at the top level. The contained JAR files are available on the classpath. However, any archives inside those JAR files are not available. The product supports one level of embedded JAR files.

Deployment of WAR files

- A composite file in a WAR file must be named `default.composite`. A composite file that is not in a WAR file can have any name.
- The `default.composite` composite file must be inside a WAR file in the `META-INF/sca-deployables` directory.
- The `META-INF/sca-deployables` directory must contain no more than one composite file. If there is more than one composite file in the `META-INF/sca-deployables` directory, then the product returns a validation error and stops the WAR file deployment.

  However, you can place other composite files in directories other than `META-INF/sca-deployables`, and reference those composite files in the top-level composite under the `META-INF/sca-deployables` directory.
- The product does not support having a `sca-contribution.xml` file inside the WAR file under the `META-INF` directory. If the product finds a `sca-contribution.xml` file, then the product returns a validation error and stops the WAR file deployment.

Notes and limitations

- The product does not provide administration console pages for configuring contributions.
- If you import a package or namepace from a different contribution, or JAR (`contribution.xml`), you might need to import that contribution as an asset before importing your own asset.

  For example, suppose your Contribution A imports a JAR file from Contribution B. You might need to import Contribution B and then Contribution A as assets. Contribution A depends on Contribution B so you must import Contribution B before importing Contribution A.
- You cannot use a local interface across a class loader boundary. Use a remotable interface to cross a class loader boundary.

Creating SCA business-level applications

You can create an empty business-level application and then add Service Component Architecture (SCA) assets, shared libraries, business-level applications, and other artifacts as composition units to the empty business-level application.

Before you begin

Configure the target application server. You must deploy SCA composite assets of a business-level application to a Version 8.x server or cluster (target) or to a Version 7.0 target that is enabled for the Feature Pack for SCA.

If your SCA composite or application uses OASIS support, you must deploy the SCA asset or application to a Version 8.5 target.

If your SCA composite or application uses Feature Pack for SCA Version 1.0.1 functionality, you must deploy the SCA asset or application to a Version 8.x target or to a feature pack Version 1.0.1.0 target. Version 1.0.1 functionality includes:

- Java Message Service (JMS) bindings
- Atom bindings
- HTTP bindings with a wire format of JSON-RPC
- Java Platform, Enterprise Edition (Java EE) integration modules (`implementation.jee`, `implementation.web`, or `implementation.ejb` components)
- SCA Spring component implementations
- OSGi applications as SCA component implementations
- Service Data Objects (SDO) composites

Optionally, determine what assets or other files that you want to add to your business-level application and whether your application files can run on your deployment targets.

About this task

You can create business-level applications using the administrative console, the wsadmin tool, or programming.

You create SCA business-level applications the same way as for non-SCA business-level applications. However, when you use an SCA asset in a business-level application, function that applies only to applications that use SCA composites becomes available. For example, you can access administrative console pages that apply only to applications that use SCA composites.

Procedure

1. Select a way to create your business-level application.

Table 275. Ways to create SCA business-level applications. You can create a business-level application using the administrative console, wsadmin scripts, or programming.

Option	Method
Administrative console business-level application creation wizard See “Creating SCA business-level applications with the console” on page 2139.	Click **Applications** > **New Application** > **New Business Level Application** and follow instructions in the wizard. For example use of the console to create a business-level application that has an SCA asset, see “Example: Creating an SCA business-level application with the console” on page 2170.

2. Create your business-level application using the administrative console, wsadmin, or programming.
3. Save the changes to your administrative configuration.

   When saving the configuration, synchronize the configuration with the nodes where the application is expected to run.

Results

The name of the application is shown in the list on the Business-level applications page.

What to do next

After you create a business-level application, you can do the following to add composition units to it:

1. Import any SCA or other assets needed by your business-level application.
2. Add assets, shared libraries, or other business-level applications as composition units.

   When you add an asset, you must specify a target that supports SCA composites. Specify only a single server or cluster as the target. Do not map an SCA composition unit to multiple servers or clusters.

   If the asset or application uses OASIS support, specify a Version 8.5 target.

   If the asset or application uses Feature Pack for SCA Version 1.0.1 functionality, specify a Version 8.x target or a feature pack Version 1.0.1.0 target.

   For applications that use `implementation.osgiapp`, add the enterprise bundle archive (EBA) asset as a composition unit to the business-level application before adding the SCA asset as a composition unit.
3. Save the changes to your administrative configuration.
4. Start the business-level application.

If the application does not run as desired, edit the application configuration, then save and run it again.

If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again. If SCA composite assets do not start, ensure that each asset is mapped to a deployment target that supports SCA composites.

When an SCA application fails to start, multiple first failure data capture (FFDC) entries are logged for a single error. The FFDC log entries pertain to the same problem and are not different issues related to the failure. Use the information provided in the FFDC log entries to fix the problem and try starting the SCA application again.

If an asset composition unit uses an Enterprise JavaBeans (EJB) binding and does not start because it has a `non-WebSphere target of "null"`, delete the asset composition unit and add it again to the business-level application. Specify a target that supports SCA composites when you add the asset to the business-level application. You cannot change the target after deployment.

If the `META-INF/sca-deployables` directory has multiple SCA composite files and the application does not start because the product `cannot obtain the CompUnitInfoLoader value`, place only the file that contains the composite in the `META-INF/sca-deployables` directory. You can place the other composite files anywhere else within the archive.

If the SCA application uses security, the target must be in the global security domain.

In multiple-node environments, synchronize the nodes after you save changes to the target before starting the business-level application.

For applications that use `implementation.osgiapp` in multiple-node environments, target the EBA composition unit to the same server or cluster as the SCA composition unit.

Creating SCA business-level applications with the console

You can create an empty business-level application and then add Service Component Architecture (SCA) assets, shared libraries, or business-level applications as composition units to the empty business-level application.

Before you begin

Configure the target application server. You must deploy SCA composite assets of a business-level application to a Version 8.x server or cluster (target) or to a Version 7.0 target that is enabled for the Feature Pack for SCA.

If your SCA composite or application uses Feature Pack for SCA Version 1.0.1 functionality, you must deploy the SCA asset or application to a Version 8.5 target or to a feature pack Version 1.0.1.0 target. Version 1.0.1 functionality includes:

- Java Message Service (JMS) bindings
- Atom bindings
- HTTP bindings with a wire format of JSON-RPC
- Java Platform, Enterprise Edition (Java EE) integration modules (`implementation.jee`, `implementation.web`, or `implementation.ejb` components)
- SCA Spring component implementations
- OSGi applications as SCA component implementations
- Service Data Objects (SDO) composites

Also, determine an application name. Optionally, determine which assets, shared libraries, or business-level applications that the new business-level application needs.

About this task

You can create a business-level application that has SCA assets using the administrative console. Alternatively, you can use the wsadmin scripting tool or programming.

You can add an asset or shared library composition unit to multiple business-level applications. However, each composition unit for the same asset must have a unique composition unit name. You can add a business-level application composition unit to more than one business-level application.

You must target an SCA composition unit to a single server or cluster, and not to multiple servers or clusters.

Procedure

1. Create an empty business-level application.
   a. Click **Applications** > **New Application** > **New Business-level Application**.
   b. On the New business-level application page, specify a unique name for the application and a description, and then click **Apply**.
   c. On the business-level application settings page, click **Save**.

   The name and description are shown in the list of applications on the Business-level applications page. Because the application is empty, its status is `Unknown`.
2. Add one SCA asset to your business-level application. The product adds the asset as a composition unit of your business-level application.
   a. Import the SCA asset.
   b. Go to the business-level application settings page.

      Click **Applications** > **Application Types** > **Business-level applications** > ***application_name***.
   c. On the business-level application settings page, specify the type of composition unit to add.

      Although you can add an asset, shared library, or business-level application to your business-level application, the logic is in your SCA asset. Add the SCA asset as a composition unit.

      Under **Deployed assets**, click **Add** > **Add Asset**.
   d. On the Add page, select one unit from the list of available units, and then click **Continue**.

On the Add page, you might be able to select multiple deployable SCA composites. However, you can deploy only one deployable SCA composite at a time. Select only one unit and click **Continue**. If you select multiple units, the product deploys only one of those units.

For applications that use `implementation.osgiapp`, add the enterprise bundle archive (EBA) asset as a composition unit to the business-level application before adding the SCA asset as a composition unit.

e. On the Set options page, change the composition unit settings as needed, and then click **Next**.

This page is not shown if you have multiple deployable unit assets.

f. On the Map composition unit to a target page, specify one target server that supports SCA composites, and then click **Next**.

The target server can be an existing cluster. To map the composition unit to a cluster, select the existing cluster from the **Available** list, click **Add**, and then click **Next**. The cluster name is shown in the **Current targets** list as `WebSphere:cluster=`*`cluster_name`*.

You must specify only a single server or cluster as the target, and not map an SCA composition unit to multiple servers or clusters.

If you are adding an SCA asset that uses security, specify a target server that is in the global security domain.

For applications that use `implementation.osgiapp` in multiple-node environments, target the EBA composition unit to the same server or cluster as the SCA composition unit.

This page is not shown when you add a business-level application.

g. On the Relationship options page, click **Next** to accept the default values.

The relationships in SCA applications are set at the asset level. Either the asset must be defined as an SCA contribution or, in the asset view, a relationship must be set to another asset. When a relationship is set to another asset manually at the asset level, the relationship only exposes all the packages within the asset to the other depended asset. The namespaces are not exposed.

This page is shown only for SCA assets that have multiple deployable or composition units.

h. On the Set Java EE composition unit relationship page, associate SCA components with Java EE applications and then click **Next**.

Java EE applications are also known as enterprise applications or enterprise archive (EAR) files. An SCA composite definition can specify an EAR file to use on the `archive` attribute of an `implementation.jee` tag. Use this page to associate SCA components in this business-level application to the EAR files named in the composite definition. If your SCA application does not use EAR files, take the default values and click **Next**.

i. On the Map security roles to users or groups page, specify security roles for users or groups as needed, and then click **Next**.

This page is only shown for SCA assets that use security.

j. On the Map RunAs roles to users page, map a user identity and password to RunAs roles as needed, and then click **Next**.

This page is only shown for SCA assets that use security.

k. On the Map virtual host page, specify a virtual host that hosts web services for each SCA composite, and then click **Next**. By default, composites map to `default_host`.

This page is only shown for SCA assets that contain a web service binding.

l. On the Attach policy set page, attach a policy set and assign policy set bindings as needed, and then click **Next**.

This page is only shown for SCA assets that use web services.

m. On the Summary page, click **Finish**. Several messages are displayed, indicating whether the product adds the unit to the business-level application successfully. A message having the format `Completed res=[WebSphere:cuname=`*`unit_name`*`]` indicates that the addition is successful. Click **Manage application**.

If the product adds the unit successfully, the name of the unit is shown in a list of deployed assets on the business-level application settings page.

If the unit addition is not successful, read the messages and add the unit again. Correct the problems noted in the messages.

n. On the Adding composition unit to the business-level application page, click **Save**.

3. Optional: Add one or more assets, shared libraries, or business-level applications to your business-level application.

   Repeat Step 2 to add another asset or add a shared library or business-level application.

Results

A business-level application that contains the specified composition units.

What to do next

After you create the application, save the changes to your configuration and start the application as needed.

If a composite asset is deployed to a target that does not support SCA composites, the SCA composite does not start. You must deploy an SCA asset to a target that supports SCA composites.

Map virtual host settings for SCA composites:

Use this page to map Service Component Architecture (SCA) composites that use a web service binding to a virtual host. You must map the composites to the virtual host that hosts the web services.

This administrative console page displays in the business-level application creation and update wizards. To view the Map virtual host page, the asset that you add to a business-level application must contain a web service binding. To view this page, do the following:

1. Import an asset that contains a web service binding.
2. Create a business-level application to which to add the asset.
3. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > **Add** > **Add Asset**.
4. On the Add composition unit page, select the asset that contains a web service binding, and click **Continue**.
5. On the Set options page, change the settings as needed and click **Next**.
6. On the Map composition unit to a target page, specify target servers as needed and click **Next**.
7. On the Define relationship with existing composition units page, change the settings as needed and click **Next**.
8. Continue changing settings as needed and click **Next** on any other pages until the Map virtual host page is displayed in the wizard.

Composite Name:

Specifies the name of the composite that uses a web service binding in the SCA artifact.

Virtual Host:

Specifies a virtual host to associate with the composite.

Select the virtual host that hosts the web services for the composite. By default, the product associates a component with the `default_host` virtual host.

Set Java EE composition unit relationships for SCA composites:

Use this page to associate Service Component Architecture (SCA) components in an SCA composite with Java Platform, Enterprise Edition (Java EE) applications, otherwise known as *enterprise applications* or *enterprise archive (EAR) files*.

An SCA composite definition can define Java EE applications as component implementations. You can define an EAR asset on an `archive` attribute of the `implementation.jee` tag for the component and use the application deployed from the asset as its implementation. On this page, associate SCA components in a business-level application with the EAR files named in the composite definition.

This administrative console page displays in the business-level application creation and update wizards. To view the Set Java EE composition unit relationship page, the asset that you add to a business-level application must contain an SCA composite. To view this page, complete the following actions:

1. Import an asset that contains an SCA composite.
2. Create a business-level application to which to add the asset.
3. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > **Add** > **Add Asset**.
4. On the Add composition unit page, select the SCA composite asset and click **Continue**.
5. On the Set options page, change the settings as needed and click **Next**.
6. On the Map composition unit to a target page, specify target servers as needed and click **Next**.
7. On the Define relationship with existing composition units page, change the settings as needed and click **Next**.
8. Continue changing settings as needed and click **Next** on any other pages until the Set Java EE composition unit relationship page is displayed in the wizard.

Component Name:

Specifies the name of an SCA component in the SCA composite that you are deploying.

EAR Asset Name:

Specifies the name of the enterprise application, or EAR file, that the SCA component uses.

An EAR asset is an EAR file that has been imported as an asset. The EAR asset name must match the `archive` attribute on the `implementation.jee` tag in the SCA composite definition.

Associated Java EE Composition Unit:

Specifies the composition unit name of the EAR asset. Select the Java EE composition unit that the SCA component uses. To associate an SCA component with an EAR file, the EAR file must be a composition unit of your SCA business-level application.

If the Java EE composition unit that you want to associate with an SCA component is not in the drop-down list, import the EAR file that is named by the `archive` attribute on the `implementation.jee` tag of the SCA composite definition as an asset. Then, add the EAR asset as a composition unit of this business-level application.

You can use the **Import an asset** and **Add an asset** links on this page to add EAR files as assets and make them composition units of your SCA business-level application.

Attach policy set settings:

Use this page to attach a policy set and assign policy set bindings for the composite defined in a Service Component Architecture (SCA) application.

This administrative console page displays in the Create new business-level application wizard. To have the Attach policy set page in the wizard, the SCA component in the asset that you add to a business-level application must use a web service binding, `binding.ws`, and the composite file or annotation must specify the intents or policy sets. To view this page, do the following:

1. Import an asset that uses a web service binding and a composite file or annotation that specifies the intents or policy sets.
2. Create a business-level application to which to add the asset.
3. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > **Add** > **Add Asset**.
4. On the Add composition unit page, select the asset that uses a web service binding, and click **Continue**.
5. On the Set options page, change the settings as needed and click **Next**.
6. On the Map composition unit to a target page, specify target servers as needed and click **Next**.
7. On the Define relationship with existing composition units page, change the settings as needed and click **Next**.
8. Continue changing settings as needed and click **Next** on any other pages until the Attach policy set page is displayed in the wizard.

To attach or detach a policy set or to assign a policy set binding, do the following:

1. Select a composite, component, service, reference, or binding from **Name**. The **Name** list is nested, indicating parent-child relationships. When you select a parent, the children are automatically selected.
2. Click the desired button.

Table 276. Button descriptions. Use the buttons to attach or detach policy sets and to assign policy set bindings.

Button	Resulting action
Attach	Attaches a policy set to the selected composite, component, service, reference, or binding. When the **Include default policy sets** option is not enabled, the options for this button contain user-created policy sets only. When the **Include default policy sets** option is not enabled and no user-created policy sets exist, then there are no button options. You can select **Include default policy sets** to display the default policy set options. When the **Include default policy sets** option is enabled, the options for this button include both default policy sets and any user-created policy sets. To attach a policy set, select a composite, component, service, reference, or binding from **Name** and click **Attach** > ***policy_set_option***. To close the menu list, click **Attach**.
Detach Policy Set	Detaches a policy set from the selected composite, component, service, reference, or binding.
Assign Service Policy Set Binding	Assigns a service policy set binding to the selected composite, component, service, reference, or binding. There are two default options: **Default** specifies to assign the default service policy set binding. **Provider Sample** specifies to assign a policy set binding that is provided with the product to the service. If you are deploying the composition unit to a server or cluster that belongs to a security domain, the list of policy set bindings consists of bindings that have been defined in the security domain to which the composition unit is being deployed.

Table 276. Button descriptions (continued). Use the buttons to attach or detach policy sets and to assign policy set bindings.

Button	Resulting action
Assign Reference Policy Set Binding	Assigns a reference policy set binding to the selected composite, component, service, reference, or binding. There are two default options: **Default** specifies to assign the default reference policy set binding. **Client Sample** specifies to assign a policy set binding that is provided with the product to the reference. If you are deploying the composition unit to a server or cluster that belongs to a security domain, the list of policy set bindings consists of bindings that have been defined in the security domain to which the composition unit is being deployed.

Include default policy sets:

Specifies whether to include default policy sets. Default policy sets specify common quality of service (QoS) behavior for generic message format.

Before selecting this option, determine whether the default policy sets provide adequate QoS characteristics for your services.

By default, this option is not enabled.

Name:

Specifies a composite, component, service, reference, or binding in the artifact.

The **Name** list is nested, indicating parent-child relationships. When you select a parent, the children are automatically selected.

Intents:

Specifies the aggregate of the intents from the composite file and the annotations. SCA intents are used to describe the abstract policy requirements of a component.

The intents shown include any intents inherited from a parent.

Matched Policy Sets:

Specifies policy sets that potentially satisfy the intents.

You can include default policy sets by enabling the **Include default policy sets** check box. To exclude default policy sets, deselect the check box.

Attached Policy Set:

Specifies attached policy sets. If no value is shown, then the composite, component, service, reference, or binding is not attached to a policy set.

To attach a policy set, select a composite, component, service, reference, or binding and click an **Attach** option.

To detach a policy set, use **Detach Policy Set**. You can detach any policy set, including pre-attached policy sets.

Policy Set Binding:

Specifies service and reference policy set bindings. If no value is shown, then the composite, component, service, reference, or binding is not assigned to a policy set binding.

To assign a policy set binding, select a composite, component, service, reference, or binding and click an **Assign Service Policy Set Binding** or **Assign Reference Policy Set Binding** option.

To reset the bindings, select the **Default** option. For example, select **Assign Service Policy Set Binding** > **Default** or **Assign Reference Policy Set Binding** > **Default**.

Map security roles to users or groups collection for SCA composites:

Use this page to view and manage mappings of security roles to users and groups that are used with the Service Component Architecture (SCA) composites.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > **Map security roles to users or groups**. This page is the same as the Map security roles to users or groups page in the Create new business-level application wizard. To view this page, your composition unit must support SCA security.

Different roles can have different security authorizations. Mapping users or groups to a role authorizes those users or groups to access applications defined by the role. Users, groups, and roles are defined when an application is installed or configured.

To map a role to a user or group, enable the Select check box beside the role name in the list and click a button. On the displayed page, specify one or more users or groups to map to the role.

Table 277. Button descriptions. Use the buttons to map security roles to users, groups, or special subjects.

Button	Resulting action
Map Users	Displays the Map users or groups page on which you can specify the users to have the selected security role.
Map Groups	Displays the Map users or groups page on which you can specify the groups to have the selected security role.
Map Special Subjects	Maps special subjects according to the option that you select: **None** specifies to map none of the special subjects to the role. **All Authenticated in Application's Realm** specifies to map all of the authenticated users to a specified role. When you map all authenticated users to a specified role, all of the valid users in the current registry who have been authenticated can access resources that are protected by this role. **All Authenticated in Trusted Realms** specifies to map all of the authenticated users in the trusted realms to a specified role. This option gives all authenticated users who belong to the user registry access to the application's realm and all authenticated users who belong to user registries access to realms which are trusted by the current security domain. **Everyone** specifies to map everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.

Role:

Specifies a security role.

Special Subjects:

Specifies which special subjects are mapped to the security role. This option applies only when an application uses multiple realms.

None Specifies to map none of the special subjects to the role.

All Authenticated in Application's Realm
Specifies to map all of the authenticated users to a specified role. When you map all authenticated users to a specified role, all of the valid users in the current registry who have been authenticated can access resources that are protected by this role.

All Authenticated in Trusted Realms
Specifies to map all of the authenticated users in the trusted realms to a specified role. All authenticated users who belong to the user registry that is mapped to the application's realm and all authenticated users who belong to user registries that are mapped to realms which are trusted by the current security domain are successfully authorized.

Everyone
Specifies to map everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.

To change the value, select the role, click **Map Special Subjects**, and select an option.

Users:

Lists the users that are mapped to the specified role within this application.

Users from the non-default realm are displayed as *user*`@`*realm*.

Groups:

Lists the groups that are mapped to this specified role within this application.

Map RunAs roles to users collection for SCA composites:

Use this page to map a specified user identity and password to a RunAs role for a Service Component Architecture (SCA) composite. This page enables you to specify application-specific privileges for individual users to run specific tasks using another user identity.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > **Map RunAs roles to users**. This page is the same as the Map RunAs roles to users page in the Create new business-level application wizard.

To view this page, the components in your composition unit must contain predefined RunAs roles and support SCA security. RunAs roles are used by components that need to run as a particular role for recognition while interacting with another component.

Username:

Specifies a user name for the RunAs role user.

This user already maps to the role specified in the Mapping users and groups to roles page. You can map the user to its appropriate role by either mapping the user to that role directly or mapping a group that contains the user to that role. After you specify the user name and password for the user and select a RunAs role, click **Apply**.

Password:

Specifies the password for the RunAs user.

Role:

Specifies a security role for a user within this application.

The authorization policy is only enforced when security is enabled.

User:

Lists the user that is mapped to the specified role within this application.

Composition unit settings

Use this page to view composition unit settings and to change the configuration properties of a composition unit. The specific settings that are available for configuration can vary, depending upon the contents of the composition unit. For example, there are additional configuration settings if the asset contained in the composition unit is an SCA composite, or an OSGi application.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_name***. The deployed asset is a composition unit of the business-level application.

- "Settings that are common to all composition units"
- "Additional composition unit settings for SCA composites" on page 2150
- "Additional composition unit settings for OSGi applications" on page 2150

Settings that are common to all composition units:

Name:

Specifies a logical name for the composition unit. You cannot change the name on this page.

Description:

Specifies a description for the composition unit.

Backing ID:

Specifies a unique identifier for a composition unit that is registered in the application management domain.

The identifier has the format `WebSphere:`*`unit_type`*`name=`*`unit_name`*. For example, for the `MyApp.jar` asset, the backing identifier might be `WebSphere:assetname=MyApp.jar`.

You cannot change the identifier on this page.

Information	Value
Data type	String
Units	Configuration unit identifier

Starting weight:

Specifies the order in which composition units are started when the server starts. The starting weight is like the startup order. The composition unit with the lowest starting weight is started first.

The value that you set for **Starting weight** determines the importance or weight of a composition unit within the business level application. For example, for the most important composition unit within a business-level application, specify 1 for **Starting weight**. For the next most important composition unit within the business-level application, specify 2 for **Starting weight**, and so on.

Note: Assign composition units upon which other composition units depend a lower starting weight than the dependent composition units. If a composition unit is not started and running before its dependent composition units, java.lang.ClassNotFoundException errors might result when you attempt to start the application or its modules.

Information	Value
Data type	Integer
Default	1
Range	0 to 2147483647

Start on distribution:

Specifies whether to start the composition unit when the product distributes the composition unit to other locations.

The default is not to start the composition unit.

This setting applies to asset or shared library composition units. This setting does not apply when the composition unit is a business-level application.

Information	Value
Data type	Boolean
Default	false

Recycle behavior on update:

Specifies whether the product restarts the composition unit after the composition unit is updated.

The default is to restart the composition unit after partial updating of the composition unit.

This setting applies to asset or shared library composition units. This setting does not apply when the composition unit is a business-level application.

Table 278. Option descriptions. Specifies whether to restart an asset or shared library composition unit.

Option	Description
ALL	Restarts the composition unit after the entire composition unit is updated
DEFAULT	Restarts the composition unit after the part of the composition unit is updated
NONE	Does not restart the composition unit after the composition unit is updated

Target mapping:

Specifies the current targets for the composition unit.

To change the deployment targets, click **Modify targets** then select a different set of deployment targets from the list of available clusters and servers.

For SCA, you must specify only a single server or cluster as the target. Do not map an SCA composition unit to multiple servers or clusters.

Note: When you change the deployment target of composition units in a business-level application, the startup order changes to the same order in which you remap composition unit targets, even if the starting weight for all composition units is set to 1. To avoid java.lang.ClassNotFoundException errors when attempting to start the remapped composition units, remap targets for composition units in the same order as that used to add the composition units or, after remapping, check starting weights to ensure that composition units upon which other composition units depend are started first.

Additional composition unit settings for SCA composites:

SCA composite components:

Specifies the component names and component implementations of SCA composites in the application.

Table 279. Column descriptions. Provides the name of each component and the name of the class or code implementing the component.

Column	Description
Component Name	Specifies the name of a component associated with the SCA composite.
Component Implementation	Specifies the name of the class or code implementing the component.

None indicates that the SCA composite does not have defined components.

SCA composite properties:

Specifies the names and values of SCA composite properties in the application.

Table 280. Column descriptions. Provides the name and value of SCA composite properties.

Column	Description
Property Name	Specifies the name of an SCA composite property.
Property Value	Specifies the value of the property.

None indicates that the SCA composite does not have defined name-value properties.

SCA composite wires:

Specifies the sources and targets of wires in the SCA composite.

Table 281. Column descriptions. Provides the source and target of wires.

Column	Description
Wire Source	Specifies the source of a wire in the SCA composite.
Wire Target	Specifies the target of the wire.

None indicates that the SCA composite does not have defined wires.

Additional composition unit settings for OSGi applications:

OSGi application deployment status:

The deployment status shows whether updates are available for the EBA asset that is contained in the composition unit. If a new version of an EBA asset is available, and all bundle downloads for the asset are complete, you can update the EBA composition unit so that the business-level application uses the latest configuration. You do not have to update the composition unit every time you update the asset.

There are four distinct deployment statuses for an EBA composition unit:

Using latest OSGi application deployment.
The composition unit is running the latest configuration of the backing asset and any CBA extensions.

New OSGi application deployment not yet available because it requires bundles that are still downloading.
The backing asset is currently undergoing a bundle version update, or bundles are downloading for a CBA extension.

New OSGi application deployment available.
The backing asset is available at a newer configuration than the configuration that is currently running in this composition unit, or a CBA extension has been added or replaced.

New OSGi application deployment cannot be applied because bundle downloads have failed.
The last bundle version update for the backing asset or CBA extension did not succeed, and therefore the newer configuration is not yet available.

If the status is "New OSGi application deployment available", the **Update to latest deployment ...** button is available. Click this button to bring the EBA composition unit up-to-date and run the updated business-level application. If any of the updates need configuration changes, a wizard prompts you to update the configuration information.

When you save the changes to the EBA composition unit, the associated business-level application is updated to use the new configuration. If the business-level application is running, the bundle and configuration updates are applied immediately. If possible (that is, depending on the nature of the updates) the system applies the updates without restarting the application. Updates that pull in new use bundles at run time prompt a full restart of the application. Updates that pull in new provision bundles might also prompt a full application restart.

JMS binding settings for SCA composites

Use this page to view property settings for a Java Message Service (JMS) binding that connects Service Component Architecture (SCA) composite references or services. Also use this page to edit resources of the JMS binding. The settings shown on this page define a `binding.jms` element.

To view this administrative console page, your composition unit must support a JMS binding reference or service. In the administrative console, do the following:

1. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name***.
2. From the composition unit settings page for SCA composites, select to view references or services:
   - For an SCA component reference, click **SCA Composite Components** > ***SCA_component_reference_name*** > **SCA Component References** > ***reference_name*** > **Bindings** > **JMS binding**.
   - For an SCA component service, click **SCA Composite Components** > ***SCA_component_service_name*** > **SCA Component Services** > ***service_name*** > **Bindings** > **JMS binding**.

You can use a JMS binding to identify existing JMS resources using Java Naming and Directory Interface (JNDI) names.

- General properties (read-only)
- Resources
- Response resources
- Request header properties for a reference (read-only)
- "Response header properties" on page 2155 for a service (read-only)

General properties:

Specifies property settings that apply to all `binding.jms` elements. The settings pertain to both services and references. You cannot edit the settings.

JMS binding URI
Specifies a uniform resource identifier (URI) that identifies properties such as the destination, connection factory and activation specification to be used to send or receive the JMS message.

The URI has the following format:

```
jms: jms_destination?
connectionFactoryName=connection_factory_name &
destinationType={queue|topic} &
deliveryMode=delivery_mode &
timeToLive=time_to_live &
priority=priority &
user_property=user_property_value & ...
```

Correlation schema
Specifies the correlation scheme used when sending reply or callback messages.

Valid values are `requestmsgidtocorrelid` (the default), `requestcorrelidtocorrelid`, and `none`.

Initial context factory
Specifies the name of the initial context factory used to obtain a JNDI initial context.

This setting is optional. If no factory is specified, the WebSphere Application Server initial context factory is used.

JNDI URL
Specifies the uniform resource locator (URL) for the JNDI provider.

Request connection
Specifies a `binding.jms` element that is present in a composite definition file.

A request is a message that is sent to an SCA service or sent by an SCA reference.

Response connection
Specifies a `binding.jms` element that is present in a composite definition file.

A response is a message received by a reference (that is, a reply from an invoked service) or a message sent by a service in response to a previous request message. A response in SCA is always a reply to a previous request.

Request wire format
Specifies the component definition element name of a wire that connects SCA composites.

For example, `wireFormat.jmsObject`.

A request is a message that is sent to an SCA service or sent by an SCA reference.

Response wire format
Specifies the component definition element name of a wire that connects SCA composites.

For example, `wireFormat.jmsObject`.

A response is a message received by a reference (that is, a reply from an invoked service) or a message sent by a service in response to a previous request message. A response in SCA is always a reply to a previous request.

Resources:

Specifies resources for the JMS binding. You can both read and edit resource settings for **Destination JNDI name**, **Activation specification JNDI name**, and **Connection factory JNDI name**. All other resource settings are read-only.

Destination type
Specifies the type of the request destination. Permitted values are `queue` (the default value) and `topic`. When `topic` is specified, then all the operations in the interface that correspond to the binding must be one-way.

This setting is for both services and references and is read-only.

`@type` is the destination type.

Destination JNDI name
Specifies an optional parameter that gives the JNDI name of a destination to which the binding is connected. For example, `jms/InvokeService_Callback`.

You can set a destination name for both services and references.

`@name` is the JNDI name of the destination.

Destination create
Specifies whether to create a destination for the binding.

Valid values are `ifnotexist` (the default), `always`, and `never`. When the value is `ifnotexist`, the product dynamically creates destination resources necessary for the SCA composite, if those resources do not exist and relate to the default messaging provider. The product creates the resources when adding the SCA composite to a business-level application.

The product validates a composite definition when adding an SCA asset to a business-level application. If the validation results in an error, the product does not add the asset to the application. If the value is `always` and the destination exists, deployment stops with an error.

Activation specification JNDI name
Specifies the activation specification that the binding uses to connect to a JMS destination to process request messages. The value must be a JNDI name.

You can set an activation specification for services only.

The attributes of this element follow those defined for the destination element.

Activation specification create
Specifies whether to create an activation specification for the binding. You can create an activation specification for services only.

Valid values are `ifnotexist` (the default), `always`, and `never`. When the value is `ifnotexist`, the product dynamically creates activation specification resources necessary for the SCA composite, if those resources do not exist and relate to the default messaging provider. If the value is `always` and the activation specification exists when the SCA composite is added to a business-level application, deployment stops with an error.

Connection factory JNDI name
Specifies the connection factory that the binding uses to process request messages. The value must be a JNDI name.

You can set a connection factory JNDI name for references only.

The attributes of this property follow those defined for the destination element. This property is mutually exclusive with the activationSpec property.

Connection factory create
Specifies whether to create a connection factory for the binding. You can create a connection factory for references only.

Valid values are `ifnotexist` (the default), `always`, and `never`. When the value is `ifnotexist`, the product dynamically creates connection factory resources necessary for the SCA composite, if those resources do not exist and relate to the default messaging provider. If the value is `always` and the connection factory exists when the SCA composite is added to a business-level application, deployment stops with an error.

Response resources:

Specifies the resources used for handling response messages, receiving responses for a reference, and for sending responses from a service. A response element defines the destination and either the connection factory or activation specification elements for handling response messages.

Response resources pertain to both services and references. You can both read and edit resource settings for **Response destination JNDI name** and **Response connection factory JNDI name**. All other resource settings are read-only.

Response destination type
Specifies the type of the response destination. Permitted values are `queue` (the default value) and `topic`. When `topic` is specified, then all the operations in the interface that corresponds to the binding must be one-way.

This setting is for both services and references and is read-only.

Response destination JNDI name
Specifies the destination that is to be used to process responses by this binding. Attributes are the same as for the parent destination element. For example, `jms/InvokeService_Response`.

You can set a destination name for both services and references.

Response destination create
Specifies whether to create a response destination for the binding.

Valid values are `ifnotexist` (the default), `always`, and `never`. When the value is `ifnotexist`, the product dynamically creates response destination resources necessary for the SCA composite, if those resources do not exist and relate to the default messaging provider. If the value is `always` and the response destination exists when the SCA composite is added to a business-level application, deployment stops with an error.

Response connection factory JNDI name
Specifies the connection factory that the binding uses to process response messages. The value must be a JNDI name.

You can set a response connection factory for both services and references.

The attributes of this element follow those defined for the destination element. This element is mutually exclusive with the activationSpec element.

Response connection factory create
Specifies whether to create a response connection factory that the binding can use to process response messages.

Valid values are `ifnotexist` (the default), `always`, and `never`. When the value is `ifnotexist`, the product dynamically creates response connection factory resources necessary for the SCA composite, if those resources do not exist and relate to the default messaging provider. If the value is `always` and the response connection factory exists when the SCA composite is added to a business-level application, deployment stops with an error.

Request header properties:

Specifies JMS header properties that apply to requests from a reference. You cannot edit the header properties.

If a JMS header property is specified, the property must not appear in the URI.

JMS type
Specifies a JMS type to use in the JMS header property using `@JMSType`.

JMS correlation ID
Specifies a JMS correlation identification to use in the JMS header property using `@JMSCorrelationID`.

JMS delivery mode
Specifies a JMS delivery mode to use in the JMS header property using `@JMSDeliveryMode`.

JMS time to live
Specifies a JMS time to live to use in the JMS header property using `@JMSTimeToLive`.

JMS priority
Specifies a JMS priority to use in the JMS header property using `@JMSPriority`.

Header property
Specifies a value to use for the specified JMS user property.

Response header properties:

Specifies JMS header properties that apply to responses from a service for outbound messages. You cannot edit the header properties.

If a JMS header property is specified, the property must not appear in the URI.

JMS type
Specifies a JMS type to use in the JMS header property using `@JMSType`.

JMS correlation ID
Specifies a JMS correlation identification to use in the JMS header property using `@JMSCorrelationID`.

JMS delivery mode
Specifies a JMS delivery mode to use in the JMS header property using `@JMSDeliveryMode`.

JMS time to live
Specifies a JMS time to live to use in the JMS header property using `@JMSTimeToLive`.

JMS priority
Specifies a JMS priority to use in the JMS header property using `@JMSPriority`.

Header property
Specifies a value to use for the specified JMS user property.

Provide HTTP endpoint URL information settings for SCA composites

Use this page to specify endpoint Universal Resource Locator (URL) prefix information for Service Component Architecture (SCA) composites accessed by web service bindings. The information is used to form complete endpoint addresses.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > **Provide HTTP endpoint URL information**.

Default SCA URL prefixes:

Shows the predefined default endpoint URL prefixes for SCA composites that are accessed by Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS) for service endpoints.

The field shows both unsecure and secure custom endpoint URL values separated by a space. For example:

```
http://theHost:9081 https://theHost:9044
```

For each endpoint URL prefix, the format is *protocol*://*host_name*:*port_number*. The protocol is either http or https. In this example, *host_name* is theHost and *port_number* is the port used in the endpoint URL.

To use the default endpoint URL prefixes, deselect the **Override default SCA URL prefixes** check box.

Override default SCA URL prefixes:

Specifies whether to use a custom endpoint URL prefix when the service has a proxied front end. The endpoint URL prefixes are those of the proxy server. You must specify proxied endpoints when deploying services that use the web service binding in a clustered configuration.

To specify a custom endpoint URL prefix, do the following:

1. Select the **Override default SCA URL prefixes** check box.
2. For **HTTP host name**, specify the host name of the unsecure custom endpoint. For example: myHost
3. For **HTTP port**, specify the port of the unsecure custom endpoint. For example: 9081
4. For **HTTPS host name**, specify the host name of the secure custom endpoint. For example: myHost
5. For **HTTPS port**, specify the port of the secure custom endpoint. For example: 9044
6. Click **OK**.

SCA composite component settings

Use this page to view and edit the attributes associated with a Service Component Architecture (SCA) component.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_name*** > ***SCA_composite_component_name***.

Components are configured instances of implementations. Components provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differently. For example each component might configure a reference of the same implementation to consume a different service.

An implementation defines the aspects configurable by a component in the form of a component type. The component type is in effect a description of the contract honored by the implementation.

A reference represents a requirement that the implementation has on a service provided by another component.

Component name:

Specifies the component name of the attribute.

Implementation:

Specifies the name of the class or configuration file that contains the component implementation.

For implementation.java, the Java class is shown. For other implementations, the name of the resource identified by the implementation is shown:

- For implementation.jee, the archive name is shown.
- For implementation.spring, the application context file is shown.
- For implementation.osgiapp, the application symbolic name and version is shown.

Type:

Specifies the type of attribute. In this case, the type is `Component`.

SCA component services:

Specifies the names of the services.

SCA component references:

Specifies the names and targets of component references. You can edit the reference target for customization.

SCA component properties:

Specifies the **Property Input Source** and **Property Value** for each property.

Options for **Property Input Source** include the following:

- **XPath** indicates the source attribute of the property.
- **File** indicates the file attribute of the property.
- **Value** indicates the property element value.

SCA component reference settings

Use this page to view and edit the attributes associated with a Service Component Architecture (SCA) component reference.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > ***SCA_composite_component_name*** > ***reference_name***.

SCA component references within an implementation represent links to services the implementation uses that must be provided by other components in the SCA system. For a composite, you can wire references of components within the composite (component references) to references of the composite (composite references), indicating that the component references must be resolved by services outside the composite.

References use bindings to describe the access methods used to invoke the services.

Under **Additional Properties**, click **View domain** to view a list of services available in the current cell or domain. This can be helpful when updating the **Target** setting value, for example.

Reference name:

Specifies the reference name of the attribute.

Type:

Specifies the type of attribute. In this case, it is `Reference`.

Reference target URI:

Specifies one or more target service uniform resource identifiers (URIs), depending on the multiplicity setting. Each target wires the reference to a component service that resolves the reference. Targets can contain a list of targets separated by a space, in the form **`target1 target2`**.

Bindings:

Specifies the URI of the binding.

Supported bindings include the SCA default binding, enterprise bean (EJB) binding, web service binding, Java Message Service (JMS) binding, Atom binding, and HTTP binding.

SCA component service settings

Use this page to view and edit the attributes associated with a component service.

To view this administrative console page, click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > ***SCA_composite_component_name*** > ***service_name***.

Services are used to publish services provided by implementations, so that they are addressable by other components.

A service published by a component can be provided by a service of a component defined within the component, or it can be provided by a component reference. The latter case allows the republication of a service with a new address or new bindings.

Service name:

Specifies the service name of the attribute.

Type:

Specifies the type of attribute. In this case, `Service`.

Work manager JNDI name:

Specifies the Java Naming and Directory Interface (JNDI) name of the work manager.

Bindings:

Specifies the uniform resource identifier (URI) of the binding.

Supported bindings include the SCA default binding, enterprise bean (EJB) binding, web service binding, Java Message Service (JMS) binding, Atom binding, and HTTP binding.

Service provider policy sets and bindings collection for SCA composites

Use this page to attach and detach policy sets to a composition unit, a service provider, its endpoints, or operations of a Service Component Architecture (SCA) composite. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service provider can share its current policy configuration.

To view this administrative console page, your composition unit must use web services and support SCA. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > **Service provider policy sets and bindings** .

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

To attach or detach a policy set or binding, do the following:

1. Select a composition unit, service, endpoint, or operation. The **Composition unit/Service/Endpoint/ Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 282. Button descriptions. Use the buttons to attach or detach policy sets and to assign policy set bindings.

Button	Resulting action
Attach	Attaches a policy set to the selected composition unit, service, endpoint, or operation. To attach a policy set, select a unit, service, endpoint, or operation and click **Attach** > ***policy_set_option***. To close the menu list, click **Attach**.
Detach Policy Set	Detaches a policy set from the selected composition unit, service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.
Assign Binding	Assigns a policy set binding to the selected composition unit, service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected service reference, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this Assign Binding action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Select this option to create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Provider sample** Select this option to use the Provider sample binding. **Provider sample V2** Select this option to use the Provider sample V2 binding when you are using either the Kerberos V5 WSSecurity default or the TrustServiceKerberosDefault policy sets. **Saml Bearer Provider sample** Select this option to use the Saml Bearer Provider sample. The Saml Bearer Provider sample extends the Provider sample binding to support SAML Bearer token usage scenarios. You can use this sample with any of the SAML bearer token default policy sets. **Saml HoK Symmetric Provider sample** Select this option to use the Saml HoK Symmetric Provider sample. The Saml HoK Symmetric Provider sample extends the Provider sample binding to support SAML holder-of-key (HoK) symmetric key token usage scenarios. You can use this sample with one of the SAML HoK Symmetric key default policy sets: either SAML11 HoK Symmetric WSSecurity default or SAML20 HoK Symmetric WSSecurity default.

Composition unit/Service/Endpoint/Operation:

Specifies the name of the composition unit and the associated service providers, endpoints or operations.

The Composition unit/Service/Endpoint/Operation column lists the composition unit and the service providers, endpoints, or operations that the composition unit contains.

Attached Policy Set:

Specifies the policy set that is attached to a composition unit, service provider, endpoint, or operation.

The Attached Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- ***Policy_set_name*** **(inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding:

Specifies the binding configuration that is available for a service provider, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- ***Binding_name*** or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- ***Binding_name*** **(inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

References policy sets and bindings collection for SCA composites

Use this page to attach and detach policy sets to a composition unit, a service reference, its endpoints, or operations of a Service Component Architecture (SCA) composite. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service reference can share its current policy configuration.

To view this administrative console page, your composition unit must use web services and support SCA. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name*** > **References policy sets and bindings**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

To attach or detach a policy set or binding, do the following:

1. Select a composition unit, service, endpoint, or operation. The **Composition unit/Service/Endpoint/ Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 283. Button descriptions. Use the buttons to attach or detach client policy sets and to assign policy set bindings.

Button	Resulting action
Attach Client Policy Set	Attaches a client policy set to the selected composition unit, service, endpoint, or operation. To attach a policy set, select a composition unit, service, endpoint, or operation and click **Attach Client Policy Set** > ***policy_set_option***. To close the menu list, click **Attach Client Policy Set**.
Detach Client Policy Set	Detaches a client policy set from the selected composition unit, service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.

Table 283. Button descriptions (continued). Use the buttons to attach or detach client policy sets and to assign policy set bindings.

Button	Resulting action
Assign Binding	Assigns a policy set binding to the selected composition unit, service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected service, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this Assign Binding action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Select this option to create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Client sample** Select this option to use the Client sample binding. **Client sample V2** Select this option to use the Client sample V2 binding when you are using either the Kerberos V5 WSSecurity default or the TrustServiceKerberosDefault policy sets. **Saml Bearer Client sample** Select this option to use the Saml Bearer Client sample. The Saml Bearer Client sample extends the Client sample binding to support SAML Bearer token usage scenarios. You can use this sample with any of the SAML bearer token default policy sets. **Saml HoK Symmetric Client sample** Select this option to use the Saml HoK Symmetric Client sample. The Saml HoK Symmetric Client sample extends the Client sample binding to support SAML holder-of-key (HoK) symmetric key token usage scenarios. You can use this sample with one of the SAML HoK Symmetric key default policy sets: either SAML11 HoK Symmetric WSSecurity default or SAML20 HoK Symmetric WSSecurity default.

Composition unit/Service/Endpoint/Operation:

Specifies the name of the composition unit and the associated service references, endpoints or operations.

The Composition unit/Service/Endpoint/Operation column lists the service composition unit and the service references, endpoints, or operations that the composition unit contains.

Attached Client Policy Set:

Specifies the policy set that is attached to a composition unit, service reference, endpoint, or operation.

The Attached Client Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- ***Policy_set_name* (inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding:

Specifies the binding configuration that is available for a service reference, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- ***Binding_name*** or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- ***Binding_name* (inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

SCA service provider settings

Use this page to manage policy sets for a Service Component Architecture (SCA) web service provider. You can attach and detach policy sets to a service provider, its endpoints, or operations. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service provider can share its current policy configuration.

To view this administrative console page, your composition unit must use web services and support SCA. Click **Services** > **Service providers** > ***service_provider_name***.

Service provider:

Specifies the full QName of the service provider. The QName must be in a format that supports the Java class `javax.xml.namespace.QName`.

For the SCA sample business-level application HelloWorldAsync, the service provider name resembles the following:

```
{http://websphere.ibm.com/HelloWorldServiceComponent/HelloWorldService}HelloWorldService
```

Policy Set Attachments:

Specifies the attached policy sets and assigned bindings for services, endpoints, or operations in the service provider.

To attach or detach a policy set or to assign bindings with system-specific configurations, do the following:

1. Select a service, endpoint, or operation. The **Service/Endpoint/Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 284. Button descriptions. Use the buttons to attach or detach policy sets and to assign policy set bindings.

Button	Resulting action
Attach	Attaches a policy set to the selected service, endpoint, or operation. To attach a policy set, select a service, endpoint, or operation and click **Attach** > ***policy_set_option***. To close the menu list, click **Attach**.
Detach Policy Set	Detaches a policy set from the selected service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.
Assign Binding	Assigns a policy set binding to the selected service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected service reference, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this Assign Binding action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Select this option to create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Provider sample** Select this option to use the Provider sample binding. **Provider sample V2** Select this option to use the Provider sample V2 binding when you are using either the Kerberos V5 WSSecurity default or the TrustServiceKerberosDefault policy sets. **Saml Bearer Provider sample** Select this option to use the Saml Bearer Provider sample. The Saml Bearer Provider sample extends the Provider sample binding to support SAML Bearer token usage scenarios. You can use this sample with any of the SAML bearer token default policy sets. **Saml HoK Symmetric Provider sample** Select this option to use the Saml HoK Symmetric Provider sample. The Saml HoK Symmetric Provider sample extends the Provider sample binding to support SAML holder-of-key (HoK) symmetric key token usage scenarios. You can use this sample with one of the SAML HoK Symmetric key default policy sets: either SAML11 HoK Symmetric WSSecurity default or SAML20 HoK Symmetric WSSecurity default.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Service/Endpoint/Operation
Specifies the name of the service and the associated service providers, endpoints or operations.

The Service/Endpoint/Operation column lists the service and the service providers, endpoints, or operations that the service contains.

Attached Policy Set
Specifies the policy set that is attached to a service provider, endpoint, or operation.

The Attached Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- ***Policy_set_name* (inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding
Specifies the binding configuration that is available for a service provider, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- ***Binding_name*** or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- ***Binding_name* (inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

About policy set bindings

In this release, there are two types of bindings: application-specific bindings and general bindings. Composition units can use both application-specific bindings and general bindings.

Application-specific bindings

You can create application-specific bindings only at a policy set attachment point. These bindings are specific to, and constrained by, the characteristics of the defined policy. Application-specific bindings can provide configuration for advanced policy requirements such as multiple signatures; however, these bindings are reusable only within an application. Also, application-specific bindings have very limited reuse across policy sets.

When you create an application-specific binding for a policy set attachment, the binding begins in a completely unconfigured state. You must add each policy, such as WS-Security or HTTP transport, that you want to override the default binding, and fully configure the bindings for each policy that you add. For WS-Security policy, some high level configuration attributes such as TokenConsumer, TokenGenerator, SigningInfo, or EncryptionInfo might be obtained from the default bindings if they are not configured in the application-specific bindings.

For service providers, you can create application-specific bindings only by selecting **Assign Binding** > **New Application Specific Binding**, on the Service providers policy sets and bindings collection page, for service provider resources that have an attached policy set. Similarly, for service clients, you can create application-specific bindings only by selecting **Assign Binding** > **New Application Specific Binding**, on the Service clients policy sets and bindings collection page, for service client resources that have an attached policy set.

General bindings

You can configure general bindings to be used across a range of policy sets and they can be reused across applications and for trust service attachments. Although general bindings are highly reusable, they cannot provide configuration for advanced policy requirements such as multiple signatures. There are two types of general bindings: general provider policy set bindings and general client policy set bindings.

You can create general provider policy set bindings by clicking **Services** > **Policy sets** > **General provider policy set bindings** > **New** in the general provider policy sets panel, or by clicking **Services** > **Policy sets > General client policy set bindings** > **New** in the general client policy set and bindings panel. For details about defining and managing service client or provider bindings, see the related links. General provider policy set bindings might also be used for trust service attachments.

SCA service client settings

Use this page to manage policy sets for a Service Component Architecture (SCA) web service client. You can attach and detach policy sets to a service reference, its endpoints, or operations. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service reference can share its current policy configuration.

To view this administrative console page, your composition unit must use web services and support SCA. Click **Services** > **Service clients** > ***service_client_name***.

Service client:

Specifies the full QName of the service client. The QName must be in a format that supports the Java class `javax.xml.namespace.QName`.

For the SCA sample business-level application HelloWorldAsync, the service client name resembles the following:

```
{http://websphere.ibm.com/HelloWorldServiceComponent/HelloWorldService}HelloWorldCallbackService
```

This SCA application has the product web service namespace, `http://websphere.ibm.com/`, and the service name in its service client name.

Policy Set Attachments:

Specifies the attached policy sets and assigned bindings for services, endpoints, or operations in the service client.

To attach or detach a policy set or to assign bindings with system-specific configurations, do the following:

1. Select a service, endpoint, or operation. The **Service/Endpoint/Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 285. Button descriptions. Use the buttons to attach or detach client policy sets and to assign policy set bindings.

Button	Resulting action
Attach Client Policy Set	Attaches a client policy set to the selected service, endpoint, or operation. To attach a policy set, select a service, endpoint, or operation and click **Attach Client Policy Set** > ***policy_set_option***. To close the menu list, click **Attach Client Policy Set**.
Detach Client Policy Set	Detaches a client policy set from the selected service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.

Table 285. Button descriptions (continued). Use the buttons to attach or detach client policy sets and to assign policy set bindings.

Button	Resulting action
Assign Binding	Assigns a policy set binding to the selected service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected service, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this Assign Binding action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Select this option to create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Client sample** Select this option to use the Client sample binding. **Client sample V2** Select this option to use the Client sample V2 binding when you are using either the Kerberos V5 WSSecurity default or the TrustServiceKerberosDefault policy sets. **Saml Bearer Client sample** Select this option to use the Saml Bearer Client sample. The Saml Bearer Client sample extends the Client sample binding to support SAML Bearer token usage scenarios. You can use this sample with any of the SAML bearer token default policy sets. **Saml HoK Symmetric Client sample** Select this option to use the Saml HoK Symmetric Client sample. The Saml HoK Symmetric Client sample extends the Client sample binding to support SAML holder-of-key (HoK) symmetric key token usage scenarios. You can use this sample with one of the SAML HoK Symmetric key default policy sets: either SAML11 HoK Symmetric WSSecurity default or SAML20 HoK Symmetric WSSecurity default.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Service/Endpoint/Operation

Specifies the name of the service and the associated service references, endpoints or operations.

The Service/Endpoint/Operation column lists the service and the service references, endpoints, or operations that the service contains.

Attached Client Policy Set

Specifies the policy set that is attached to a service reference, endpoint, or operation.

The Attached Client Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- ***Policy_set_name* (inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding

Specifies the binding configuration that is available for a service reference, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- ***Binding_name*** or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- ***Binding_name* (inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

About policy set bindings

In this release, there are two types of bindings: application-specific bindings and general bindings. Composition units can use both application-specific bindings and general bindings.

Application-specific bindings

You can create application-specific bindings only at a policy set attachment point. These bindings are specific to, and constrained by, the characteristics of the defined policy. Application-specific bindings can provide configuration for advanced policy requirements such as multiple signatures; however, these bindings are reusable only within an application. Also, application-specific bindings have very limited reuse across policy sets.

When you create an application-specific binding for a policy set attachment, the binding begins in a completely unconfigured state. You must add each policy, such as WS-Security or HTTP transport, that you want to override the default binding, and fully configure the bindings for each policy that you add. For WS-Security policy, some high level configuration attributes such as TokenConsumer, TokenGenerator, SigningInfo, or EncryptionInfo might be obtained from the default bindings if they are not configured in the application-specific bindings.

For service providers, you can create application-specific bindings only by selecting **Assign Binding** > **New Application Specific Binding**, on the Service providers policy sets and bindings collection page, for service provider resources that have an attached policy set. Similarly, for service clients, you can create application-specific bindings only by selecting **Assign Binding** > **New Application Specific Binding**, on the Service clients policy sets and bindings collection page, for service client resources that have an attached policy set.

General bindings

You can configure general bindings to be used across a range of policy sets and they can be reused across applications and for trust service attachments. Although general bindings are highly reusable, they

cannot provide configuration for advanced policy requirements such as multiple signatures. There are two types of general bindings: general provider policy set bindings and general client policy set bindings.

You can create general provider policy set bindings by clicking **Services** > **Policy sets** > **General provider policy set bindings** > **New** in the general provider policy sets panel, or by clicking **Services** > **Policy sets** > **General client policy set bindings** > **New** in the general client policy set and bindings panel. For details about defining and managing service client or provider bindings, see the related links. General provider policy set bindings might also be used for trust service attachments.

Example: Creating an SCA business-level application with the console

You can add many different types of artifacts to business-level applications. For example, you can add applications or modules, Java archives (JAR files), data in compressed files, and other business-level applications. This example describes how to create an empty business-level application and then add a Service Component Architecture (SCA) JAR file to the application using the administrative console.

Before you begin

In a product installation, verify that the target server is configured. As part of configuring the server, determine whether your application files can run on your deployment target. You must deploy SCA composite assets of a business-level application to a Version 8.0 or later server or cluster (target) or to a Version 7.0 target that is enabled for the Feature Pack for SCA.

Download the `helloworld-ws-asynch.jar` SCA sample file from a product download site:

1. Go to the **Samples, Version 8.5** information center.
2. On the **Downloads** tab, click **FTP** or **HTTP** in the **Service Component Architecture** section.
3. In the authentication window, click **OK**.
4. In the **SCA.zip** compressed file, go to the `SCA/installableApps` directory and download the `helloworld-ws-asynch.jar` file.

About this task

For this example, use the administrative console to create a business-level application named HelloWorldAsync that has an SCA JAR file, `helloworld-ws-asynch.jar`, as an asset.

Procedure

1. Create an empty business-level application named `HelloWorldAsync`.
   a. Click **Applications** > **New Application** > **New Business Level Application**.
   b. On the New application page, specify the name `HelloWorldAsync`, optionally add a description, and then click **Apply**.
   c. On the page that is displayed, click the **Save** link.

   The name is shown in the list of applications on the Business-level applications page. Because the application is empty, its status is `Unknown`.
2. Import the SCA JAR asset.
   a. Click **Applications** > **New Application** > **New Asset** in the console navigation tree.
   b. On the Upload asset page, specify the asset package to import, `helloworld-ws-asynch.jar`, and click **Next**.

      The JAR file is in the *app_server_root*/`installableApps` directory.
   c. On the Select options for importing an asset page, click **Next** to accept the default values.
   d. On the Summary page, click **Finish**.
   e. On the Adding asset to repository page, if messages show that the operation completed, click **Manage assets**.
   f. On the Assets page, click the **Save** link.

The file name displays in the list of assets.

3. Add the SCA JAR asset as a composition unit of the business-level application.
   a. Click **Applications** > **Application Types** > **Business-level applications**.
   b. On the Business-level applications page, click the `HelloWorldAsync` application name.
   c. On the business-level application settings page, click **Add** > **Add Asset**.
   d. On the Add page, select the `helloworld-ws-asynch.jar` asset composition unit from the list of available units, and then click **Continue**.
   e. On the Set options page, click **Next** to accept the default values.
   f. On the Map composition unit to a target page, specify a target server that supports SCA composites, and then click **Next**.

      The target server can be an existing cluster. To map the composition unit to a cluster, select the existing cluster from the **Available** list, click **Add**, and then click **Next**. The cluster name is shown in the **Current targets** list as `WebSphere:cluster=`*`cluster_name`*.
   g. On the Define relationship with existing composition units page, click **Next** to accept the default values.
   h. On the Map virtual host page, click **Next** to accept the default values.
   i. On the Summary page, click **Finish**.

      Several messages are displayed. A message having the format `Completed res=[WebSphere:cuname=helloworldws]` indicates that the addition is successful.

      During deployment of the composition unit, you can view the Uniform Resource Identifier (URI) for composite level service of some bindings, along with the service name and binding type. Only the URI is editable. The product does not validate the URI.
   j. If the addition is successful, click **Manage application**.
   k. On the business-level application settings page, click **Save**.

   The asset name and type displays in the list of deployed assets. If you click on the asset name, the composition unit settings page displays, with the asset name in the **SCA Composite Components** list.
4. Start the HelloWorldAsync business-level application.
   a. Click **Applications** > **Application Types** > **Business-level applications**.
   b. On the Business-level applications page, select the check box beside HelloWorldAsync.
   c. Click **Start**.

   When the business-level application is running, a green arrow displays for **Status**. If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again.

What to do next

Optionally examine, and possibly use in applications, other SCA sample files in the `SCA/installableApps` directory of the downloadable SCA samples.

If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again. If SCA composite assets do not start, ensure that each asset is mapped to a deployment target that supports SCA composites.

If an asset composition unit uses an Enterprise JavaBeans (EJB) binding and does not start because it has a `non-WebSphere target of "null"`, delete the asset composition unit and add it again to the business-level application. Specify a target that supports SCA composites when you add the asset to the business-level application. You cannot change the target after deployment.

If the SCA application uses security, the target server or cluster must be in the global security domain.

In multiple-node environments, synchronize the nodes after you save changes to the target before starting the business-level application.

Updating SCA composite artifacts

You can view and update Service Component Architecture (SCA) composite artifacts in business-level applications.

Before you begin

Add an SCA artifact as a composition unit to a business-level application.

About this task

You can view and update the following SCA composite artifacts:
- Composite level property definition
- Composite level component property definition
- Composite level component reference definition

You can view and update SCA composite artifacts using the administrative console or the wsadmin tool. This topic describes how to view and update SCA composite artifacts using the administrative console.

Procedure

1. Go to the composition unit settings page for an SCA composite artifact in a business-level application.

   Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***SCA_deployed_asset_composition_unit_name***.

   The composition unit settings page for an SCA composite artifact has fields that are not shown on the composition unit settings page for a non-SCA artifact:
   - SCA composite components
   - SCA composite properties
   - SCA composite wires
2. Click on a name link in one of these SCA fields to view the settings for an SCA artifact.

   The SCA fields display `None` instead of a name link if the composition unit does not have that particular type of SCA composite.
3. Optional: Update a SCA composite setting value.
   a. Change an existing setting value for the SCA artifact.
   b. Click **OK**.

   The setting value is updated.

Viewing and updating SCA composites in HelloWorldAsync

"Example: Creating an SCA business-level application with the console" on page 2170 describes how to create the HelloWorldAsync business-level application. This application contains an SCA artifact, helloworldws, as a composition unit. You can view and update settings for SCA composites in the helloworldws composition unit using the console.

1. Go to the composition unit settings page for the helloworldws composition unit in the HelloWorldAsync business-level application.

   Click **Applications** > **Application Types** > **Business-level applications** > **HelloWorldAsync** > **helloworldws**.

   From the composition unit settings page, you can view information associated with helloworldws, as well as update composite settings.
2. Click on a link for the SCA artifact to be viewed or updated.

For example, click on the HelloWorldServiceComponent link under **SCA composite components** and, in the page that displays, click on the HelloWorldService link under **Service**. In the Component service settings page that displays, you can specify a setting value for the service.

3. If you update a setting value for the SCA artifact, click **OK**.

What to do next

Save the changes to your administrative configuration.

On multiple-server products, when saving the configuration, synchronize the configuration with the nodes where the application is expected to run.

Viewing SCA composite definitions

You can view information on the definition of a Service Component Architecture (SCA) composite in the administrative console.

Before you begin

The SCA composite must be a composition unit in a business-level application.

About this task

The composite definition provides data on the composite, such as component names and service references. The View composite page displays the composite definition of an SCA deployed asset composition unit.

Procedure

1. Go to the View composite page.

   Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***SCA_deployed_asset_name*** > **View composite**.

2. Optional: Click **Expand All** or **Collapse All** to more easily browse the page.

Results

The View composite page displays the contents of the composition unit definition.

Example

Suppose the HelloWorldAsync business-level application provided as a sample with the product is installed. Click **Applications** > **Application Types** > **Business-level applications** > **HelloWorldAsync** > **helloworldws** > **View composite**.

The View composite page displays configuration information resembling the following:

```
<composite targetNamespace="http://helloworld" name="helloworldws" >
  <component name="AsynchTranslatorComponent" >
    <implementation.java class="helloworld.impl.AsynchTranslatorComponent" />
      <service name="AsynchTranslatorService">
         <interface.java interface="helloworld.AsynchTranslatorService"
              callbackInterface="helloworld.HelloWorldCallback" />
        <binding.ws/>
          <callback>
              <binding.ws/>
          </callback>
      </service>
  </component>
</composite>
```

What to do next

Browse the page to ensure that it contains the intended configuration information.

Viewing SCA domain information

You can view information on Service Component Architecture (SCA) composites in an SCA domain in the administrative console.

Before you begin

The SCA composite must be a composition unit in a business-level application.

About this task

Viewing SCA domain information enables you to see on one console page information on all components in an SCA domain. The View domain page displays information on available services in the current domain.

Procedure

1. Go to the View domain page.

   Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***SCA_deployed_asset_name*** > **View domain**.
2. Optional: Click **Expand All** or **Collapse All** to more easily browse the page.

Results

The View domain page lists information on components in the current domain.

Example

Suppose the HelloWorldAsync business-level application provided as a sample with the product is installed. Click **Applications** > **Application Types** > **Business-level applications** > **HelloWorldAsync** > **helloworldws** > **View domain**.

The View domain page displays information resembling the following:

```
<domain name="myCell02">
  <component name = "AsynchTranslatorComponent"
        mapTarget = "WebSphere:cell=myCell02,node=myNode02,server=server1">
    <service name = "AsynchTranslatorService">
      <interface.java interface = "helloworld.AsynchTranslatorService"/>
    </service>
    <reference name = "AsynchTranslatorService" target = ""/>
    <httpurlendpoints name = "endpoints" uri = ""/>
  </component>
</domain>
```

What to do next

Browse the page to ensure that it contains the intended information.

You can export the same domain information to a file using the exportCompositeToDomain command. See “Exporting SCA domain information using scripting.”

Viewing and editing JMS bindings on references and services of SCA composites

You can view information on a Java Message Service (JMS) binding for a Service Component Architecture (SCA) composite in the administrative console. The JMS bindings page of the console displays the settings of a `binding.jms` element. You can use the console page to edit resource and response resource settings.

Before you begin

Configure an SCA composite that uses JMS bindings and add that composite as a composition unit to a business-level application. To view and edit settings for a JMS binding on references, the composite must define a JMS binding in the reference. Similarly, to view and edit settings for a JMS binding on services, the composite must define a JMS binding in the service.

About this task

The JMS bindings page enables you to see JMS binding settings for an SCA composite in the administrative console. The information shown is similar to that shown by running the **viewCompUnit** wsadmin scripting command to view the SCA composite composition unit. However, the console page shows JMS binding settings only, and does not show other information on the composition unit that running **viewCompUnit** returns.

After deployment of an SCA composite that uses JMS bindings, you can edit JMS binding resource or response resource settings that specify Java Naming and Directory Interface (JNDI) names on the JMS bindings page. The editing capabilities are similar to those of the **editCompUnit** scripting command. To edit settings, the JMS resource must exist. The product does not dynamically create JMS resources when you edit a composition unit.

Procedure

1. Go to the JMS bindings page.

   To view this page, your composition unit must support a JMS binding reference or service.

   a. Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***deployed_asset_composition_unit_name***.

   b. From the composition unit settings page for SCA composites, select to view references or services:

      - For an SCA component reference, click **SCA Composite Components** > ***SCA_component_reference_name*** > **SCA Component References** > ***reference_name*** > **Bindings** > **JMS binding**.
      - For an SCA component service, click **SCA Composite Components** > ***SCA_component_service_name*** > **SCA Component Services** > ***service_name*** > **Bindings** > **JMS binding**.

2. View the JMS binding settings for the composition unit.
3. Edit the **Destination JNDI name**, **Activation specification JNDI name**, **Connection factory JNDI name**, **Response destination JNDI name**, or **Response connection factory JNDI name** settings as needed and click **Apply**.

Results

The JMS bindings page displays property settings for a JMS binding. If you change resource or response resource JNDI name values, the changed values are shown.

Example

Suppose that you have a business-level application named `MyJmsBLA`, which has an SCA composite named `MySCAComposite`. This composite uses a JMS binding on service named `JmsService`, in a composition unit named `myJmsBindingCU`.

1. Click **Applications** > **Application Types** > **Business-level applications** > **MyJmsBLA** > **myJmsBindingCU** > **SCA Composite Components** > **MySCAComposite** > **SCA Component Services** > **JmsService** > **Bindings** > **JMS binding**.

   The JMS bindings page displays general binding settings and settings for a JMS binding on services.
2. Edit the value specified for **Activation specification JNDI name**, and click **Apply**.

   The JMS bindings page displays the changed value.

What to do next

Browse the page to ensure that it contains the binding information.

For information on JMS binding settings, refer to the online help for the JMS bindings page or Section 1.4 of the SCA JMS Binding specification, Version 1.00.

Exporting WSDL and XSD documents

You can export Web Services Description Language (WSDL) and XML schema definition (XSD) documents used by a Service Component Architecture (SCA) composition unit to a location of your choice.

Before you begin

Your SCA business-level application must contain one or more composition units that use a WSDL or XSD document.

A WSDL document is a file that provides a set of definitions that describe a web service in WSDL, an Extensible Markup Language (XML)-based description language.

An XSD document is an instance of an XML schema written in the XML schema definition language. The document has the extension `.xsd`. The prefix `xsd` in the XML elements of an XSD document indicates the XML schema namespace.

About this task

You can export WSDL and XSD documents that are used by an SCA composition unit using the administrative console. In previous releases, you had to use the **`exportWSDLArtifacts`** command to export WSDL and XSD documents.

On the composition unit settings page for an SCA composite, click the **Export WSDL and XSD documents** link and then specify the target location for the files.

The product extracts from the selected composition unit the WSDL and XSD files that are required for web services client development. The files are for the services exposed by the web service binding, `binding.ws`.

Procedure

1. Go to the composition unit settings page for the SCA composite.

   Click **Applications** > **Application Types** > **Business-level applications** > ***application_name*** > ***SCA_deployed_asset_name***.
2. Click the **Export WSDL and XSD documents** link.

3. From the displayed dialog, specify the target directory to which to save the documents.

Results

The WSDL and XSD documents are copied to the target directory.

Example

Suppose you want to export WSDL or XSD documents in the HelloWorldAsync business-level application that the product provides as a sample. Complete the following actions in the administrative console:

1. Click **Applications** > **Application Types** > **Business-level applications** > **HelloWorldAsync** > **helloworldws** > **Export WSDL and XSD documents**.
2. Using the displayed dialog, specify a directory that exists on your computer.

The product adds the `helloworldws_WSDLArtifacts.zip` file to the specified directory. The `helloworldws_WSDLArtifacts.zip` file has one WSDL file, `AsynchTranslatorComponentTranslatorService_wsdlgen.wsdl`.

What to do next

Examine the exported files to ensure that they contain the intended WSDL and XSD documents.

You can export WSDL and XSD documents using the exportWSDLArtifacts command. See Exporting WSDL and XSD documents using scripting.

Deploying OSGi applications that use SCA

You can deploy OSGi and Service Component Architecture (SCA) applications to servers or clusters as assets, which you add to a business-level application by creating composition units.

Before you begin

This topic assumes that you have already created an OSGi application packaged as an enterprise bundle archive (EBA) file and an SCA composite definition packaged as a Java archive (JAR) file.

An SCA composite packaged with a Web archive (WAR) file cannot use the `implementation.osgiapp` component.

About this task

To deploy an OSGi application that uses SCA, create a business-level application that includes the EBA and SCA assets. You can use the administrative console or wsadmin commands to create the business-level application, import the EBA file and SCA composite as assets, and then add the EBA and SCA assets as composition units to the business-level application.

An EBA composition unit consists of the imported asset (the EBA file), plus any configuration information for the OSGi application context roots, virtual hosts, and resource bindings. If a deployed OSGi application is to be extended using composite bundles extensions, apply all extensions to the EBA composition unit (`cu`) and update the OSGi application to the latest deployment version. You must install all service import and exports that are provided by composite bundle extensions before you can add the SCA asset as a composition unit to a business-level application. For information on using and deploying OSGi composite bundle extensions, see the topic on extending a deployed OSGi application.

For `implementation.osgiapp`, the component implementation value for the composition unit is the application symbolic name and version.

The following deployment restrictions apply to EBA files:
- Import an EBA file into only one asset.
- Add an EBA asset to only one business-level application.

For information about these and other restrictions, see the topic on multiple SCA implementation packaging considerations.

Procedure

1. Create a business-level application.

Table 286. Ways to create SCA business-level applications. You can create a business-level application that has OSGi and SCA assets using the administrative console or wsadmin scripts.

Option	Method
Administrative console business-level application creation wizard	Click **Applications** > **New Application** > **New Business Level Application** and follow instructions in the wizard. For example use of the console to create a business-level application that has an SCA asset, see Example: Creating an SCA business-level application with the console.

2. Import OSGi, SCA, or other assets needed by your business-level application.
3. Add the EBA asset as a composition unit to the business-level application.

   You must add the EBA asset before the SCA asset that references it.
4. Add the SCA asset as a composition unit to the business-level application.

   By default, the product assigns the SCA composition unit a higher starting weight value than the value for the OSGi composition unit. When using `implementation.osgiapp`, the SCA composition unit must start after the referenced OSGi composition unit starts. Thus, the SCA composition unit must have a higher starting weight value than the OSGi composition unit. If you override the default values and set a higher starting weight value for the OSGi composition unit than for the SCA composition unit, the business-level application does not start.

   In a Network Deployment environment, target the OSGi composition unit to the same server as the SCA composition unit.

   Note: If a validation error occurs when adding the SCA asset, saying that a service is not found, ensure that the following conditions exist:
   - The service name matches the name of a blueprint service
   - The Blueprint service uses the `services.exported.interfaces` property to export its interface
   - The interface is included in the Application-ExportService header of the application manifest

   The absence of these conditions can cause the validation error. The error resembles the following:

```
com.ibm.wsspi.management.bla.op.OpExecutionException: CWSAM0105E:
The following Service Component Architecture (SCA) Validation errors caused the
CreateScaCodeGen step to fail:
  Service not found for component service:
   Component = HelloWorldComponent Service = helloWorld
```

Results

The business-level application has OSGi and SCA composition units. You can now start the business-level application.

If the ServiceRuntimeException `Unable to access OSGi application framework` occurs when starting the business-level application, ensure that the SCA composition unit has a higher starting weight than the EBA composition unit.

If the ServiceRuntimeException `Unable to obtain service service from application` occurs, check `SystemOut.log` and `SystemErr.log` files for messages from the Blueprint container indicating that the run time cannot start the Blueprint service.

If the java.lang.TypeNotPresentException occurs when starting the business-level application, a bundle that uses one of the supported SCA annotations is not importing the org.osoa.sca.annotations package. Update the bundle manifest to import this package.

Note: This topic references one or more of the application server log files. As a recommended alternative, you can configure the server to use the High Performance Extensible Logging (HPEL) log and trace infrastructure instead of using `SystemOut.log` , `SystemErr.log`, `trace.log`, and `activity.log` files on distributed and IBM i systems. You can also use HPEL in conjunction with your native z/OS logging facilities. If you are using HPEL, you can access all of your log and trace information using the LogViewer command-line tool from your server profile bin directory. See the information about using HPEL to troubleshoot applications for more information on using HPEL.

What to do next

If you must delete a composition unit from the business-level application, delete the SCA composition unit before attempting to delete the EBA composition unit, otherwise the product returns an error. An EBA composition unit that provides the implementation cannot be deleted until the SCA composition unit that uses the `implementation.osgiapp` is deleted.

Multiple SCA implementation packaging considerations

The product supports several implementation technologies that can provide the business logic for SCA service components. Because a primary architectural objective of SCA is to enable you to combine existing services together, it is easy to assume that a single composite with multiple components might have different implementation technologies. However, there are restrictions that limit packaging of multiple SCA implementations.

Business-level application deployment constraints

Before deploying an SCA application, particularly one that uses an OSGi application, consider the following constraints:

- Collocation

  SCA composites can be comprised of a variety of implementation frameworks. Packaging formats for these frameworks include, for example, enterprise archive (EAR) files for Java Platform, Enterprise Edition (Java EE) applications or enterprise bundle archive (EBA) files for OSGi applications. An SCA business-level application must contain both the framework packaging format and the SCA composite that uses it.

  In a Network Deployment environment, when deploying an `implementation.osgiapp` in an SCA composition unit, the EBA composition unit used for the implementation must be in the same business-level application and targeted to the same server or cluster as the SCA composition unit.
- Artifacts or shared libraries

  Business-level applications support shared libraries. It typically is good practice to package interface artifacts (Java interface classes or WSDL/XSD artifacts) in a shared library. Services in applications configured in a business-level application can reference the shared library.

  However, OSGi applications do not support shared configuration libraries. You must package Java interface and WSDL/XSD artifacts in both the SCA contribution Java archive (JAR) file and the EBA asset that is configured in the business-level application.
- Only one deployment

  Only one instance of a given OSGi application, packaged as an EBA file, can be deployed in a Network Deployment cell. An EBA file can contain only one OSGi application. An EBA asset can contain only one

EBA file. An EBA asset can be added to only one business-level application, which is scoped at cell-level. Therefore, you can specify only once that a given OSGi application be an SCA implementation.

To get around this constraint, you can connect to a deployed OSGi application from other applications in the cell, but you cannot include the business logic in another composite.

Class loading

A purpose of the OSGi application framework is to build a class loader structure that is wholly defined by the OSGi service registry and OSGi application design, and avoid class loader problems that affect Java EE servers.

If your applications use different implementation technologies, differences in class loader structures might cause the SCA run time to copy objects from one environment to another. OSGi applications are remote-only interfaces that do not permit pass-by-reference behavior.

Your SCA application must use correct semantics for local and remote interfaces.

Chapter 46. Deploying SIP applications

Use the administrative console to customize your Session Initiation Protocol (SIP) application installation

About this task

When you deploy a Session Initiation Protocol (SIP) application, you can perform various tasks such as installing, starting, stopping, upgrading, and uninstalling the application.

SIP applications are installed as Java Platform, Enterprise Edition (Java EE) applications. You can deploy a SIP application from a graphical interface or from a command line.

Deploying SIP applications through the console

You can deploy a Session Initiation Protocol (SIP) application through the administrative console.

Before you begin

SIP applications are deployed as Java 2 Platform Enterprise Edition (J2EE) applications. In order to process requests, a virtual host must be defined when deploying the SIP application. If there is no virtual host defined for the configured SIP container listen port, the installed application will be inaccessible.

Procedure

1. Open the administrative console.

   In a browser, go to URL `http://`*`hostname`*`:9090/admin`, where *hostname* is the name of the host computer. Enter the appropriate login information, and click **OK**.
2. In the left frame click **Applications** > **Install New Application**.
3. Browse and select a SAR file. Specify the context root, beginning with a slash (/), in the **Context Root** field. For example, if your application is named ThisApplication, type `/ThisApplication`.
4. Click **Next** (under the **Context Root** field not beside the WebSphere Status title). If the SAR file has been assembled correctly, the screen will still have the title “Preparing for the application installation”, but the content will change. If an error message appears, check the contents of the SAR file; in particular, verify the `web.xml` file contents, and try to reload the SAR file.
5. Click **Next**. If you see a screen indicating “Application Security Warnings”, click **Continue**.
6. The **Install New Application** screen should appear with “Step 1: Select application options” highlighted. Select the options you need and click **Next**.
7. “Step 2: Map modules to servers” should appear highlighted now. You can choose the cluster or server where you want to install the application's modules.
   - If you are installing the application in a stand-alone system, click **Next**.
   - If you are installing the application in a clustered system, select **`WebSphere:cell=`*`cellname`*`,cluster=`*`cluster_name`** in the **Clusters and Servers** field, select the check box beside the web module that you want to install, and click **Apply** and **Next**.
8. Now “Step 3: Map virtual hosts for web modules” should appear highlighted. To the right of the application name there should be a drop-down labeled **Virtual Host**.
   - If you are installing the application in a stand-alone system, set the value of the drop-down to **default_host**, and click **Next**.
   - If you are installing the application in a clustered system, set the value of the drop-down to the name of the virtual host that was chosen during setup, and click **Next**.

   Remember: You must define a virtual host for your configured SIP container listen port or else you will not be able to access the application.

9. You should now see “Step 4: Summary” highlighted. In the right panel you will see a **Summary of installation options** table that details your selected options and their values. If you need to change an option, click **Previous** to return to the section where you can make your change. Click **Finish** to install the application with your settings. The screen should display, `Application` *appname*`_sar installed successfully`, where *appname* is the name of the application.
10. Click the **Save to Master Configuration** link. A Save to Master Configuration window appears.
11. In the Save to Master Configuration window, click **Save**. The application has now been saved in the current configuration.
12. To confirm that the installation succeeded, in the left frame click **Applications** > **Enterprise Applications**. The newly installed application should appear in the list of installed applications as *appname*_sar.
13. To start the application so that it can service SIP requests, check the box beside *appname*_sar, and click **Start**. You might also want to look at the logs for a successful startup message.

Results

The application can service SIP requests now.

Deploying SIP applications through scripting

You can deploy a Session Initiation Protocol (SIP) application not only from the administrative console but also from a command line.

About this task

Note: To deploy a SIP application, the application must exist with an enterprise archive (EAR) file, a Session Initiation Protocol (SIP) module (SAR file), or a web application archive (WAR) file.

Use the wsadmin scripting tool to deploy applications from a command line.

Procedure

- Launch a scripting client.

  For more information, see AdminApp object for scripted administration.
- List applications.

  For more information, see Listing applications using the wsadmin scripting tool.
- Install stand-alone archive files.

  For more information about installation, see Installing enterprise applications using wsadmin scripting and Installation options for the AdminApp object.
- Edit application configurations.

  For more information, see Editing application configurations using the wsadmin scripting tool.
- Uninstall applications.

  For more information, see Uninstalling enterprise applications using the wsadmin scripting tool.

Upgrading SIP applications

Follow these steps to upgrade SIP applications.

About this task

Use the following application upgrade procedure to install a new version of a previously installed application without any outage.

Procedure

1. Install version 1 of the application in cluster 1.
2. Create cluster 2.
3. Set cluster 1 as the default cluster.
4. Install version 2 of the application in cluster 2.
5. Change the default cluster at proxy to cluster 2.
6. To shut down Version 1, monitor PMI and watch for the application session for this application to come to zero. When it is at zero or a level you consider okay to lose, stop the application.

Results

New calls will be routed to cluster 2. Old calls will continue to be routed to cluster 1.

Chapter 47. Deploying web applications

This page provides a starting point for finding information about web applications, which are comprised of one or more related files that you can manage as a unit, including:

- HTML files
- Servlets can support dynamic web page content, provide database access, serve multiple clients at one time, and filter data.
- Java ServerPages (JSP) files enable the separation of the HTML code from the business logic in web pages.

IBM extensions to the JSP specification make it easy for HTML authors to add the power of Java technology to web pages, without being experts in Java programming.

Deploying JavaServer Pages and JavaServer Faces files

JSP class loading settings

You can configure a JavaServer Pages (JSP) class to be loaded by either the JSP engine's class loader or by the web module's class loader.

By default, a JSP class is loaded by a unique instance of the JSP engine's class loader. The JSP engine's class loader enables reloading at runtime of a JSP class when the JSP source or one of its dependents is modified. This allows you to reload a single JSP class when necessary, without affecting any other loaded JSP classes.

JSP classes are loaded by the web module's class loader under either of the following scenarios.

1. The JSP engine configuration parameter useFullPackageNames is set to true, and the JSP file is configured as a servlet in the `web.xml` file using the *<servlet-class>* scenario in the table later in this topic.
2. The JSP engine configuration parameters useFullPackageNames and disableJspRuntimeCompilation are both set to true. In this case, you do not need to configure a JSP file does as a servlet in the `web.xml` file.

Configuring JSP files as Servlets

You can configure a JSP file as a servlet in the `web.xml` file. There are two ways to do this. They are described in the table later in this section.

Before you configure a JSP file as a servlet, consider the following.

1. Reloading capability - If runtime reloading of JavaServer Pages files is desired, requests for JavaServer Pages files must be handled by the JSP engine. The *<servlet-class>* scenario in the table later in this section disables runtime JSP file reloading, while the *<jsp-file>* scenario is compatible with reloading.
2. Reducing the number of class loaders - If you do not require runtime reloading of modified JSP pages and you want to reduce the number of class loader instances, then you can use the *<servlet-class>* scenario in the table that follows. Similarly, scenario 2 in section 1 can be used without having to configure a JSP file as a servlet.

Table 287. Example: Configure a JSP file as a servlet in the web.xml file.. Configure a JSP file as a servlet

Scenario	Example	compatible with runtime reloading	multiple class loaders used?	useFullPackageNames
<jsp-file>	<servlet> <servlet-name>jspOne</servlet-name> <jsp-file>jspOne.jsp</jsp-file> </servlet>	Yes	Yes	Can be true or false
<servlet-class>	<servlet> <servlet-name>jspTwo</servlet-name> <servlet-class>_ibmjsp.jspTwo</servlet-class> </servlet>	No	No	Must be true

The JSP batch compiler tool helps you configure JavaServer Pages files as servlets. When useFullPackageNames is true, the JSP batch compiler generates *<servlet>* and *<servlet-mapping>* elements for each JSP file that it successfully translates and compiles. The elements are written to a `web.xml` fragment file named `generated_web.xml` which is located in the binaries `WEB-INF` directory of a web module processed by the JSP file batch compiler (this directory is located within the deployed application's ear file). You can copy and paste all or some of these elements into the `web.xml` file to configure JavaServer Pages files as servlets.

Take note of the location of the `web.xml` that is used by the application server. The application specific configuration is obtained from either the application binaries (the application's ear file) or from the configuration repository. If an application is deployed into WebSphere Application Server with the flag Use Binary Configuration set to true, then the `WEB-INF/web.xml` file is looked for in a web module's binaries directory, not in the configuration repository. Examples of these two locations follow:

- An example of a configuration repository directory is `{WAS_ROOT}/profiles/profilename/config/cells/`*`cellname`*`/applications/`*`enterpriseappname`*`/deployments/`*`deployedname/webmodulename`*
- An example of an application binaries directory is: `{WAS_ROOT}/profiles/`*`profilename`*`/installedApps/`*`nodename`*`/EnterpriseAppName/`*`WebModuleName/`*

If the JSP batch compiler is executed on a pre-deployed application then the `web.xml` file is in the web module's `WEB-INF` directory.

JavaServer Pages (JSP) runtime reloading settings

JavaServer Pages files can be translated and compiled at run time when the JSP file or its dependencies are modified. This is known as JSP reloading.

Note: Use an assembly tool, such as Rational Application Developer, to modify IBM extension and binding files. You can convert extension and binding files within modules from XMI to XML using the IBM Bindings and Extensions Conversion Tool for Multi-Platforms.

JSP reloading is enabled through the reloadEnabled JSP engine parameter in the `WEB-INF/ibm-web-ext.xmi` or `WEB-INF/ibm-web-ext.xml` file.

`ibm-web-ext.xmi` example:

```
<jspAttributes xmi:id="JSPAttribute_1" name="reloadEnabled" value="true"/>
```

`ibm-web-ext.xml` example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-ext
   xmlns="http://websphere.ibm.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
   xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-web-ext_1_0.xsd"
      version="1.0">

   <jsp-attribute name="trackDependencies" value="true" />
   <jsp-attribute name="disableJspRuntimeCompilation" value="true" />
   <jsp-attribute name="reloadEnabled" value="true"/>

   <reload-interval value="5"/>
   <auto-encode-requests value="false"/>
   <auto-encode-responses value="false"/>
   <enable-directory-browsing value="false"/>
   <enable-file-serving value="false"/>
   <pre-compile-jsps value="false"/>
   <enable-reloading value="true"/>
   <enable-serving-servlets-by-class-name value="false" />
</web-ext>
```

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

The following table contains suggested reload settings for production and development environments.

Table 288. Suggested reload settings for production and development environments.. Reload settings

Configuration Attribute	Production Environment setting	Development Environment setting
reloadEnabled	false	true
reloadInterval	n/a (ignored if reloadEnabled is false)	approximately 5 seconds
trackDependencies	n/a (ignored if reloadEnabled is false)	true Alternatively, set this to false to improve response time if dependencies are not changing
disableJspRuntimeCompilation	true - Alternatively, set this to false if JSP files are not pre-compiled and therefore need to be compiled on the first request.	false

The default for the reloadEnabled parameter is true. If the reloadEnabled parameter is set to true, a JSP file is reloaded at run time if the JSP file and its class file do not have the same timestamp. In addition, if trackDependencies is set to true then the JSP file is reloaded if the timestamp of any of its dependencies has changed since the JSP class file was last generated. If the reloadEnabled parameter is set to false, a JSP file is still compiled if necessary on the first request to it unless the parameter disableJspRuntimeCompilation is true. For example, when disableJspRuntimeCompilation is false and reloadEnabled is false, a JSP file is compiled on the first request if the class file is outdated. It would not compile on subsequent requests, even if the JSP source file is modified or the class file is deleted,, unless reloadEnabled is true.

Reload interval

The reload interval is set through the reloadInterval JSP engine parameter.

`ibm-web-ext.xmi` example:

```
<jspAttributes xmi:id=JSPAttribute_1 name=reloadInterval value=5/>
```

ibm-web-ext.xml example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-ext
   xmlns="http://websphere.ibm.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-web-ext_1_0.xsd"
      version="1.0">

   <jsp-attribute name="trackDependencies" value="true" />
   <jsp-attribute name="disableJspRuntimeCompilation" value="true" />
   <jsp-attribute name="reloadInterval" value="5"/>

   <reload-interval value="5"/>
   <auto-encode-requests value="false"/>
   <auto-encode-responses value="false"/>
   <enable-directory-browsing value="false"/>
   <enable-file-serving value="false"/>
   <pre-compile-jsps value="false"/>
   <enable-reloading value="true"/>
   <enable-serving-servlets-by-class-name value="false" />
</web-ext>
```

If reloading is enabled, the reloadInterval parameter value determines the delay between checks to see if a JSP file is outdated. For example, if reloadInterval is 5, the JSP engine checks to see if a JSP file is outdated only when the last such check was done more than five seconds prior to the current request for the JSP file. Once the reloadInterval is exceeded, reload checking is performed and the reload interval timer is reset to 0 for that JSP file. The larger the reloadInterval, the less frequently the JSP engine checks for the need to reload a JSP file.

Dependency tracking

Dependency tracking is set through the trackDependencies JSP engine parameter.

ibm-web-ext.xmi example:

```
<jspAttributes xmi:id="JSPAttribute_1" name="trackDependencies" value="true"/>
```

ibm-web-ext.xmi example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-ext
   xmlns="http://websphere.ibm.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-web-ext_1_0.xsd"
      version="1.0">

   <jsp-attribute name="trackDependencies" value="true" />
   <jsp-attribute name="disableJspRuntimeCompilation" value="true" />
   <jsp-attribute name="reloadInterval" value="5"/>

   <reload-interval value="5"/>
   <auto-encode-requests value="false"/>
   <auto-encode-responses value="false"/>
   <enable-directory-browsing value="false"/>
   <enable-file-serving value="false"/>
   <pre-compile-jsps value="false"/>
   <enable-reloading value="true"/>
   <enable-serving-servlets-by-class-name value="false" />
</web-ext>
```

If reloading is enabled, the trackDependencies parameter value determines whether the JSP engine tracks modifications to the requested JSP file dependencies as well as to the JSP file itself. The three types of dependencies tracked by the JSP engine are:

- files statically included in the JSP file
- tag files that are referenced in the JSP file (excluding tag files that are in JAR files)
- TLDs that are referenced in the JSP file (excluding TLDs that are in JAR files)

Dependency tracking information is always included in the generated class file even if trackDependencies is false. The information is not used by the JSP engine or batch compiler unless the trackDependencies parameter is true. This means that you can enable dependency tracking without having to recompile JSP files.

For example, the toplevel.jsp file statically includes the footer.jspf file. When the toplevel.jsp file is compiled, the path to the footer.jspf file and its timestamp are stored in the toplevel.jsp's class file. As a result, the footer.jspf file is modified and the toplevel.jsp file is requested. Now that the reload interval for the toplevel.jsp file has been exceeded, the JSP engine compares the timestamp stored in the class file with the footer.jspf file timestamp on disk. Because the timestamps are different, the toplevel.jsp file is compiled, picking up the modification to the footer.jspf file. In order for dependency tracking to work, the trackDependencies value must be set to true at the time a JSP file is requested at run time or is processed by the batch compiler.

Disabling compilation

Disablement of run time compilation of JavaServer Pages is set via the disableJspRuntimeCompilation JSP engine parameter.

ibm-web-ext.xmi example:

```
<jspAttributes xmi:id="JSPAttribute_1" name="disableJspRuntimeCompilation" value="true"/>
```

ibm-web-ext.xml example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-ext
   xmlns="http://websphere.ibm.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee http://websphere.ibm.com/xml/ns/javaee/ibm-web-ext_1_0.xsd"
      version=1.0>

   <jsp-attribute name="trackDependencies" value="true" />
   <jsp-attribute name="disableJspRuntimeCompilation" value="true" />
   <jsp-attribute name="reloadInterval" value="5"/>

   <reload-interval value="5"/>
   <auto-encode-requests value="false"/>
   <auto-encode-responses value="false"/>
   <enable-directory-browsing value="false"/>
   <enable-file-serving value="false"/>
   <pre-compile-jsps value="false"/>
   <enable-reloading value="true"/>
   <enable-serving-servlets-by-class-name value="false" />
</web-ext>
```

If the disableJspRuntimeCompilation parameter is set to true, the JSP engine at run time does not translate and compile JSP files; the JSP engine loads only precompiled class files. JSP source files do not need to be present in order for the class files to be loaded. With this option set to true, an application can be installed without JSP source, but must have precompiled class files. There is a web container custom property of the same name that can be used to determine the behavior of all web modules installed in a server. If both the web container custom property and the JSP engine option are set, the JSP engine option takes precedence. Setting the disableJspRuntimeCompilation parameter to true automatically sets reloadEnabled to false.

Reload processing sequence

The processing sequence pertaining to JSP file reloading when trackDependencies is false is shown in Figure 1.

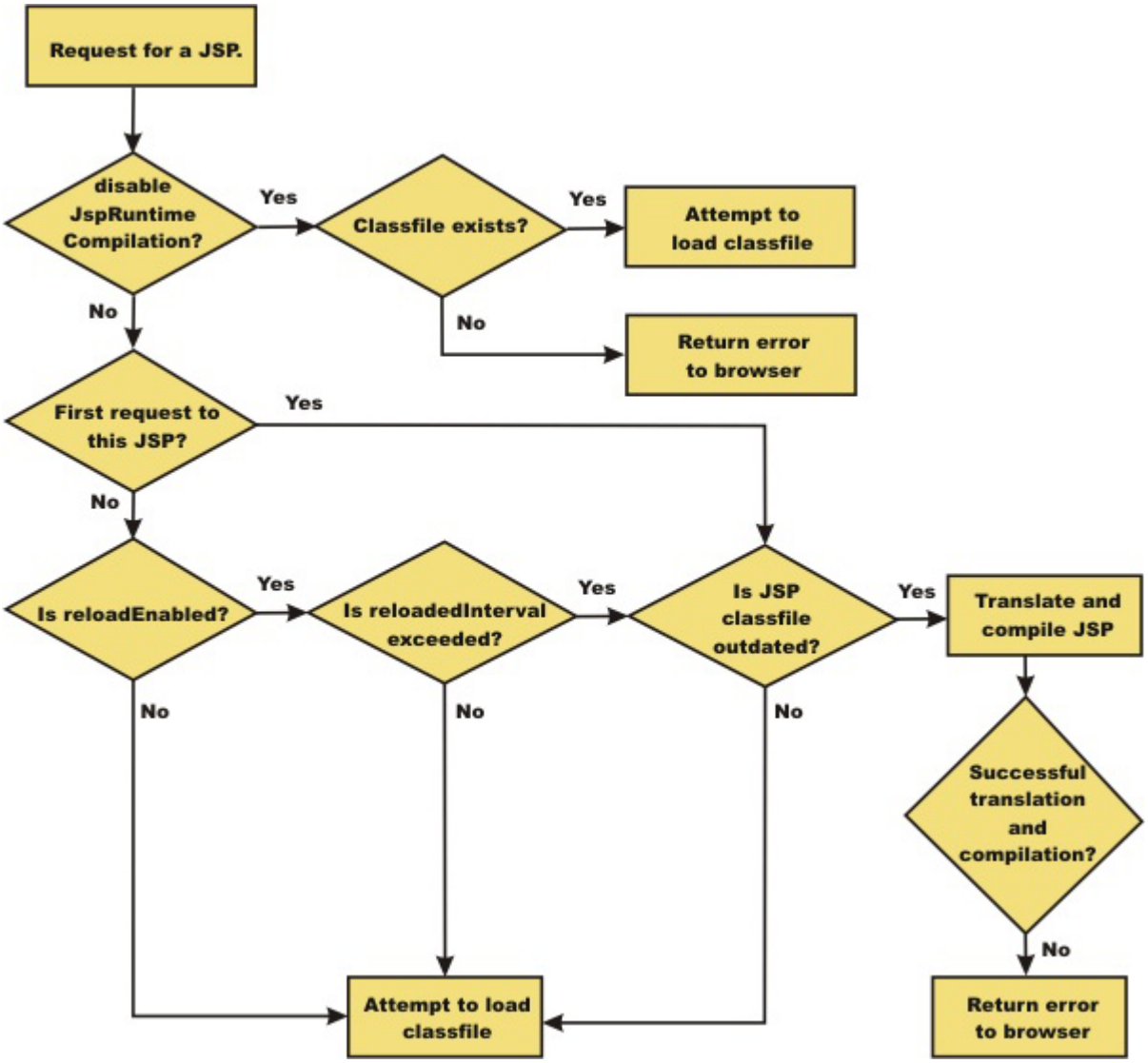


Figure 9. Reload processing sequence when trackDependencies is false.

When trackDependencies is true, the JSP engine does additional file system processing to determine if any of a JSP file's dependencies have changed since the JSP file was last translated and compiled. Figure 2 shows the additional processes that are performed on the 'No' path of flow chart labeled “is JSP class file outdated?”. You can see that the path taken when disableJspRuntimeCompilation is true is the most efficient path.

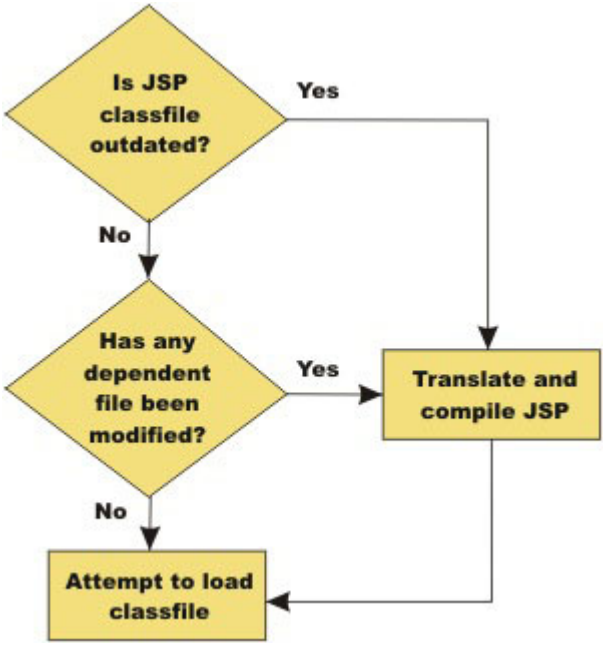


Figure 10. Additional reload processing performed when trackDependencies is true.

JSP and JSF option settings

Use this page to configure the class reloading of web modules such as JavaServer Pages (JSP) files and to select a JSF implementation to use with this application.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **JSP and JSF options**. This page is the same as the **Provide JSP reloading options for web modules** page on the application installation and update wizards.

The following note applies to the files with an `.xmi` extension in this topic:

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

Web module

Specifies the name of a web module in the installed or deployed application.

URI
Specifies the location of the module that is relative to the root of the application (EAR file).

JSP enable class reloading
Specifies whether to enable class reloading when JSP files are updated.

A web container reloads JSP files only when the IBM extension reloadEnabled in the jspAttributes of the `ibm-web-ext.xmi` file is set to `true`.

Java Platform, Enterprise Edition 5 (Java EE 5) and later applications IBM extension files are in `.xml` file format. For applications versions earlier than Java EE 5, they are in the `.xmi` file format.

JSP reload interval in seconds
Specifies the number of seconds to scan the application file system for updated JSP files. The default is the value of the reloading interval attribute in the IBM extension (`META-INF/ibm-web-ext.xmi`) file of the web module.

To enable reloading, specify a value greater than zero (for example, 1 to 2147483647). The default reload interval is 5. To disable reloading, specify zero (0). The range is from 0 to 2147483647.

The reloading interval attribute takes effect only if class reloading is enabled.

Java EE 5 applications and later IBM extension files are in `.xml` file format. For applications versions earlier than Java EE 5, they are in the `.xmi` file format.

Sun Reference Implementation 1.2
Select this option to use the Sun Reference Implementation 1.2 JSF implementation.

If you change the JSF implementation that you are using for your application, you must delete any previously compiled JSP files. If you precompiled your application, you must recompile. If you did not precompile, but have already requested JSP files from this application, you must delete the JSP files from the temp directory of your profile.

You can set the JSF engine configuration parameter, com.ibm.ws.jsf.JSF_IMPL_CHECK, to true to automatically mark the JSP files to recompile at application startup.

MyFaces 2.0
Select this option to use the MyFaces JSF implementation. This is the default JSF implementation.

If you change the JSF implementation that you are using for your application, you must delete any previously compiled JSP files. If you precompiled your application, you must recompile. If you did not precompile, but have already requested JSP files from this application, you must delete the JSP files from the temp directory of your profile.

You can set the JSF engine configuration parameter, com.ibm.ws.jsf.JSF_IMPL_CHECK, to true to automatically mark the JSP files to recompile at application startup.

In a mixed-version cell, a Version 7 node uses MyFaces 1.2 if the MyFaces selection is toggled, while a Version 8 and later node uses MyFaces 2.0. For WebSphere Application Server versions before Version 7 (for example, Version 6.1 and earlier), this toggle is ineffective because JSF implementation switching was not supported before Version 7.

JSP run time compilation settings

By default, the JavaServer Pages (JSP) engine translates a requested JSP file, compiles the `.java` file, and loads the compiled servlet into the run time environment. You can change the JSP engine default behavior by indicating that a JSP file must not be translated or compiled at run time, even when a `.class` file does not exist.

If run time compilation is disabled, you must precompile the JSP files, which provides the following advantages:

- Reduces compilation related disk operations.
- Minimizes disk storage requirements necessary for handling temporary `.java` files generated during a run time compilation.
- Allows you to not include the JSP source files in the application.
- Allows verification that a JSP file compiled successfully before deploying and installing the application in the product

You can disable run time JSP file compilation on a global or an individual web application basis:

- To disable the translation and compilation of JSP files for all Web applications, in the administrative console, click **Servers > Server Types > WebSphere application servers >** *server_name*. Then, in the Container Settings section, click **Web container settings > Web container > Custom properties**.

  If the disableJspRuntimeCompilation property appears in the list of defined custom properties, but is set to `false`, click the property name, and then set the property to `true`.

  If this property is not included in the list of defined custom properties, click **New**, and then specify `disableJspRuntimeCompilation` in the **Name** field and `true` in the **Value** field.

  Valid settings for this property are `true` or `false`. When this property is set to `true`, translation and compilation of the JSP files is disabled at run time for all web applications.
- To disable the translation and compilation of JSP files for a specific web application, set the JSP engine initialization parameter disableJspRuntimeCompilation to `true`. This setting, if enabled, determines the run time behavior of the JSP engine and overrides the web container custom property setting.

  Set this parameter through the **JavaServer Pages attribute assembly settings** page when assembling applications.

  Valid values for this setting are `true` or `false`. If this parameter is set to `true`, then, for that specific web application, translation and compilation of the JSP files is disabled at run time, and the JSP engine only loads precompiled files.
- If neither the web container custom property nor the JSP parameter is set, the first request for a JSP file results in the translation and compilation of the JSP file when the `.class` file does not exist or is outdated. Subsequent requests for the file also result in translations and compilations, but only if the following conditions are met:
  - Translations are required because the `.class` file is outdated.
  - Reloading is enabled for the web module.
  - Reload interval is exceeded.

If you disable run time compilation and a request arrives for a JSP file that does not have a matching .class file, the JSP engine returns the following 404 error to the browser:

```
Error 404: SRVE0200E: Servlet [org.apache.jsp._jsp1]: Could not find required servlet class - _jsp1.class
```

In this case, an exception is written to the joblog (SYSPRINT) file if the ras_trace_outputLocation property in the was.env file is set to SYSPRINT, or to CTRACE if ras_trace_outputLocation is set to BUFFER.

If a JSP file has a matching .class file but that file is out of date, the JSP engine still loads the .class file into memory.

Provide options to compile JavaServer Pages settings

Use this page to specify options to be used by the JavaServer Pages (JSP) compiler.

This administrative console page is a step in the application installation and update wizards. To view this page, you must select **Precompile JavaServer Pages files** on the **Select installations options** page. Thus, to view this page, click **Applications** > **New Application** > **New Enterprise Application** > ***application_path*** > **Next** > **Detailed - Show me all installation options and parameters** > **Next** > **Next** or **Continue** > **Precompile JavaServer Pages files** > **Next** > **Step: Provide options to compile JSPs**.

You can specify the JSP compiler options on this page only when installing or updating an application that contains web modules. After the application is installed, you must edit the JSP engine configuration parameters of a web module `WEB-INF/ibm-web-ext.xmi` file to change its JSP compiler options.

Note: For IBM extension and binding files, the .xmi or .xml file name extension is different depending on whether you are using a pre-Java EE 5 application or module or a Java EE 5 or later application or module. An IBM extension or binding file is named ibm-*-ext.xmi or ibm-*-bnd.xmi where * is the type of extension or binding file such as app, application, ejb-jar, or web. The following conditions apply:

- For an application or module that uses a Java EE version prior to version 5, the file extension must be .xmi.
- For an application or module that uses Java EE 5 or later, the file extension must be .xml. If .xmi files are included with the application or module, the product ignores the .xmi files.

However, a Java EE 5 or later module can exist within an application that includes pre-Java EE 5 files and uses the .xmi file name extension.

The `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `ibm-webservicesclient-bnd.xmi`, `ibm-webservicesclient-ext.xmi`, and `ibm-portlet-ext.xmi` files continue to use the .xmi file extensions.

Web module
Specifies the name of a module within the application.

URI
Specifies the location of the module relative to the root of the application (EAR file).

JSP class path
Specifies a temporary class path for the JSP compiler to use when compiling JSP files during application installation. This class path is not saved when the application installation is complete and is not used when the application is running. This class path is used only to identify resources outside of the application which are necessary for JSP compilation and which will be identified by other means (such as shared libraries) after the application is installed. In network deployment configurations, this class path is specific to the deployment manager machine.

To specify that multiple web modules use the same class path:
1. In the list of web modules, select the **Select** check box beside each web module that you want to use a particular class path.
2. Expand **Apply Multiple Mappings**.
3. Specify the desired class path.
4. Click **Apply**.

Use full package names
Specifies whether the JSP engine generates and loads JSP classes using full package names.

When full package names are used, precompiled JSP class files can be configured as servlets in the `web.xml` file, without having to use the `jsp-file` attribute. When full package names are not used, all JSP classes are generated in the same package, which has the benefit of smaller file-system paths.

When the options useFullPackageNames and disableJspRuntimeCompilation are both `true`, a single class loader is used to load all JSP classes, even if the JSP files are not configured as servlets in the `web.xml` file.

This option is the same as the useFullPackageNames JSP engine parameter.

JDK source level

Specifies the source level at which the Java compiler compiles JSP Java sources. Valid values are `13`, `14`, and `15`. The default value is `13` for pre-Java EE 5 web modules, which specifies source level 1.3 and `15` for Java EE 5 and later web modules.

Disable JSP runtime compilation

Specifies whether a JSP file should never be translated or compiled at run time, even when a `.class` file does not exist.

When this option is set to `true`, the JSP engine does not translate and compile JSP files at run time; the JSP engine loads only precompiled class files. JSP source files do not need to be present in order to load class files. You can install an application without JSP source, but the application must have precompiled class files.

For a single web application class loader to load all JSP classes, this compiler option and the **Use full package names** option both must be set to `true`.

This option is the same as the disableJspRuntimeCompilation JSP engine parameter.

Deploying web applications using RRD

Task overview: Assembling applications using remote request dispatcher

Remote request dispatcher (RRD) is a pluggable extension to the web container which allows application frameworks, servlets and JavaServer Pages (JSP) to include content from outside the currently executing resource's Java Virtual Machine (JVM) as part of the response sent to the client.

Before you begin

You must have WebSphere Application Server, Network Deployment installed to use remote request dispatcher function. You should also familiarize yourself the limitations of remote request dispatcher. See article, Remote request dispatcher considerations for details.

Procedure

1. Install enterprise application files with the console.
2. Configure the sending of include requests between the application and remote resources.
   - Configure web applications to dispatch remote includes.
   - Configure web applications to service remote includes.
3. Optional: Modify your application to locate resources located in two different contexts using the servlet programming model.

   The Servlet Programming Model for including resources remotely does not require you to use any non-Java Platform, Enterprise Edition (Java EE) Servlet Application Programming Interfaces (APIs). The remote request dispatcher (RRD) component follows the same rules to obtain a ServletContext and a remote resource. By using JavaServer Pages standard tag library (JSTL), your application is further removed from obtaining a ServletContext object or RequestDispatcher that is required in the framework example in the following step because the JSTL custom tag does this implicitly. Study the following example of a sample JavaServer Pages application to learn how to locate resources that are in two different contexts, investments and banking.

```
<HEAD>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8059-1"
isELIgnored="false"
%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" $>
</HEAD>
<BODY>

<%--

Programming example using JavaServer Pages and JavaSever Pages
Standard Tag Library (JSTL).
JSTL provides a custom tag to import contents (in servlet and JSP
terms include) in the scope of the same request from outside of
the current web module context by specifying a context parameter.

JSTL restriction: The web module that is imported
must run inside of the same JVM as the calling resource
if imported URL is not fully qualified.

RRD extends this functionality by permitting the web module to
be located within the scope of the current WebSphere Application Server
core group versus the scope of the JVM.
--%>


<hr size="5"/>
<%--  Include resource investmentSummary.jsp located in the
  web application with context root of /investments. --%>

<c:import url="investmentSummary.jsp" context="/investments"/>

<hr size="5"/>
<%--  Include resource accountSummary.jsp located in the
  web application with context root of /banking. --%>

<c:import url="accountSummary.jsp" context="/banking"/>

<hr size="5"/>

</BODY>
</HTML>
```

4. Optional: Modify your application to locate resources located in two different contexts using the framework programming model.

   The Framework Programming Model for including resources remotely does not require you to use any non- Java Platform, Enterprise Edition (Java EE) Servlet Application Programming Interfaces (APIs). When a request is initiated for a ServletContext name that is not presently running inside of the current web container, the remote request dispatcher (RRD) component returns a ServletContext object that can locate a resource that exists anywhere inside a WebSphere Application Sever WebSphere Application Server, Network Deployment environment provided that the resource exists and RRD is enabled for that ServletContext object. Study the following sample framework snippet that demonstrates how to locate resources located in two different contexts, investments and banking.

```
/*
Programming example using a generic framework.
Servlet Specification provides an API to obtain
a servlet context in the scope of the same request
different from the current web module context by
specifying a context parameter.

Servlet Specification restriction: The web module that obtain
must run inside of the same JVM as the calling resource.

RRD extends this functionality by permitting the web module to be located
within the scope of the current WebSphere Application Server core group
versus the scope of the JVM.

*/

protected void frameworkCall (ServletContext context, HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException(

   PrintWriter writer = response.getWriter();

   writer.write("<HTML>");
   writer.write("<HEAD>");
   writer.write("</HEAD>");
   writer.write("<BODY>");
   writer.write("<hr size=\"5/">);

   //Include resource investmentSummary.jsp located in web application
```

```
   //with context root of /investments.
   RequestDispatcher rd = getRequestDispatcher ( context, "/investments", "/investmentSummary.jsp");
   rd.include(request, response);

   writer.write("<hr size=\"5/">);

   //Include resource accountSummary.jsp located in web application
   //with context root of /banking.
   rd = getRequestDispatcher ( context, "/banking", "/accountSummary.jsp");
   rd.include(request, response);

   writer.write("</BODY>");
   writer.write("</HTML>");
}
private RequestDispatcher getRequestDispatcher (ServletContext context, String contextName, String resource) {
   return context.getContext(contexName).getRequestDispatcher(resource);
}
```

Results

After enabling at least one enterprise application to dispatch remote includes and at least one enterprise application to service remote includes, RRD is now enabled.

What to do next

Restart the modified applications if already installed or start newly installed applications to enable RRD on each application.

Remote request dispatcher:

Remote Request Dispatcher (RRD) is a pluggable extension to the web container for application frameworks, servlets, and JavaServer Pages to include content from outside of the current Java virtual machine (JVM) for the resource, as part of the response sent to the client.

Remote request dispatcher is an extensible infrastructure for other components and stack products to add custom extensions like generators and handlers, to the RRD extension. The remote request dispatcher extension enhances the standard Java Platform, Enterprise Edition (Java EE) `javax.servlet.RequestDispatcher` implementation to be aware of locating remote resources using web services to communicate between machines within a WebSphere Application Server, Network Deployment (ND) core group. The remote request dispatcher extension reports any errors that occur on the remote server back to the originating server. It can also use SSL for secure communications and WS-Security security context propagation between servers. See the rrdSecurity.props file topic for more information.

RRD portlet support carries forward the remote request dispatcher concept to portlets and enhances the portlet container for invocation of portlets outside of the current JVM resource.

By using the RRD extension, you can share request load across multiple machines and JVMs by including remote servers within the cell. If RRD resource is memory or processor intensive, the calling resource is not affected as much as a standard RequestDispatcher running within the same JVM. RRD solves this problem by separating resources into a different JVM.

Capabilities

- Requests on remote server are treated as include requests. Filters and request listeners are started as if the dispatch type is INCLUDE.
- Serializable request attributes and query parameters are sent to remote server.
- Security context is sent to a remote server through LTPA tokens.
- Servlet parameters and OutputStream

  Request parameters are passed to remote server.
- Response headers that are set by the remotely included resource are ignored similar to includes on a local server. Internal headers such as Set-Cookie can still be set and are propagated back.
- All original request headers are passed to remote server

  - Similar to the plug-in for WebSphere Application Server.
  - Method calls return the state as if they are on local server. For example, getServer returns the local server name or isSecure returns whether the request to the ‘local' server has been secure.
- Cookies and sessions
  - Cookies are passed to the remote server as part of headers.
  - Sessions in local and remote servers use the same cookie or session ID for a given client which is similar to includes in the same server. If a session exists on a remote server, the session cookie contains the information for both the servers to maintain the affinity to the remote server.
- Exceptions
  - If there is an exception on the remote server, the server returns an RRD-specific web services fault which wraps the original exception created by the application.
  - Attempt to recreate the original exception on the local server if the exception class exists on both servers. If the original exception cannot be recreated, an RRD-specific ServletException is constructed and used instead.
  - The exception is recreated by the local server for error handling purposes.
- Dynamic cache

  When dynamic cache is enabled, caching is performed on the local and remote machine.
- Security

  You can use SSL to encrypt RRD messages between application servers. SSL is enabled by default, however, you must also pass security context needs through RRD to ensure that the security state is available in the remote machine. RRD uses WS-Security to pass this information, but this security context propagation is disabled by default. See the rrdSecurity.props file topic for additional information.

Chapter 48. Deploying web services

Deploying web services applications onto application servers

After assembling the artifacts required to enable the web module for web services into an enterprise archive (EAR) file, you can deploy the EAR file into the application server.

Before you begin

To deploy Java-based web services, you need an enterprise application, also known as an EAR file that is configured and enabled for web services.

A Java API for XML-Based Web Services (JAX-WS) application does not require additional bindings and deployment descriptors for deployment whereas a Java API for XML-based RPC (JAX-RPC) web services application requires you to add additional bindings and deployment descriptors for application deployment. JAX-WS is much more dynamic, and does not require any of the static data generated by the deployment step required for deploying JAX-RPC applications.

For JAX-WS web services, the use of the `webservices.xml` deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

Note: In a mixed node cell, you can only target a JAX-WS enabled enterprise beans module to a server using WebSphere Application Server Version 7.0 and later. However, you can target a JAX-WS enabled web application archives (WAR) module to a server using either WebSphere Application Server Version 7.0 and later or WebSphere Application Server Version 6.1 Feature Pack for Web Services

You can use the **wsdeploy** command with JAX-RPC applications to add WebSphere product-specific deployment classes to a web services-compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file.

To install or deploy a JAX-WS application, you only need to install the JAX-WS enabled EAR file. If your web services application contains only JAX-WS endpoints, you do not need to run the **wsdeploy** command, as this command is used to process only JAX-RPC endpoints.

Ensure that you have installed the HTTP or Java Message Service (JMS) router module that was generated with the **endptEnabler** command onto the same target as your web services enterprise bean JAR files. These HTTP or JMS router modules are included in your web services application and they need to use the runtime libraries of the application server.

About this task

This task is one of the steps in developing and implementing web services.

You can use either the administrative console or the **wsadmin** scripting tool to deploy an EAR file. If you are installing an application containing web services by using the **wsadmin** command, specify the `-deployws` option for JAX-RPC applications. If you are installing an application containing web services by using the administrative console, select **Deploy WebServices** in the Install New Application wizard. For more information about installing applications using the administrative console, see the installing enterprise application files with the console information.

If your JAX-RPC web services application was previously deployed with the **wsdeploy** command, it is not necessary to specify web services deployment during installation.

The following actions deploy the EAR file with the **wsadmin** command:

Procedure

1. Start *install_root*/bin/wsadmin from a command prompt.
2. Deploy the EAR file.
   - For JAX-WS web service applications, enter the `$AdminApp install` *EARfile* `"-usedefaultbindings"` command at the **wsadmin** prompt.
   - For JAX-RPC web service applications, enter the `$AdminApp install` *EARfile* `"-usedefaultbindings -deployws"` command at the **wsadmin** prompt.

Results

You have a web service installed onto your application server.

Note: While installing web services applications that contain a large number of enterprise beans onto the application server, you might receive out of memory errors. If you receive out of memory errors, increase the heap size of your Java Virtual Machine (JVM). If you are installing the application server in a network deployment environment, you might need to increase the heap size of the JVM in the application servers in which you are installing the application, and in the deployment manager profile, dmgr. Read about tuning the IBM virtual machine for Java documentation to learn more about tuning the application server environment.

What to do next

You can confirm that the web services application was deployed by entering the web service endpoint URL in a browser, then viewing an informative page. The information page contains the following information:

```
{http://webservice.pli.tc.wssvt.ibm.com}RetireWebServices
Hello! This is an Axis2 web service!
```

The first line of this information is variable, depending on your web service. The URI in the brackets is the namespace and the string that follows, in this example RetireWebServices, is the name of the port used to access the web service.

The next step you might want to consider is to apply security to your web service.

Provide options to perform the web services deployment settings

Use this page to specify options for web services deployment.

This administrative console page is a step in the application installation and update wizards.

To view this page, you must select **Deploy web services** on the **Select installation options** page.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **New application** > ***application_path*** .
2. Select the option to **Show all installation options and parameters** .
3. Click **Next** to get to the **Step: Select installation options** page.
4. Select **Deploy web service**.
5. Click **Next** to get to the **Step: Provide options to perform the web services deployment** page.

You can specify the web services deployment options on this page only when installing or updating an application that uses web services.

The **wsdeploy** command is supported by Java API for XML-based RPC (JAX-RPC) applications. The Java API for XML-Based Web Services (JAX-WS) programming model that is implemented by the application server does not support the **wsdeploy** command. If your web services application contains only JAX-WS endpoints, you do not need to run the **wsdeploy** command, as this command is used to process only JAX-RPC endpoints.

The options that you specify set parameter values for the **wsdeploy** command. The **wsdeploy** command adds product-specific deployment classes to a web services-compatible enterprise archive (EAR) file or an application client Java archive (JAR) file. These classes include:
- Stubs
- Serializers and deserializers
- Implementations of service interfaces

The **wsdeploy** command is run during installation after you click **Finish** on the **Summary** page of the wizard.

Deploy web services option - Classpath
Specifies entries to add to the CLASSPATH when the generated classes are compiled.

To specify the class paths of multiple entries, you need to separate the entries with a semicolon on Windows platforms and on Linux, Unix, and z/OS platforms, you need to use a colon to separate the entries. This is the same separator that is used with the CLASSPATH environment variable.

This option is the same as the **wsdeploy** command parameter `-cp` *`class_path`*.

Information	Value
Data type	String
Default	null

Deploy web services option - Extension Directories
Specifies a directory that contains zipped or Java archive (JAR) files. All zipped and JAR files in this directory are added to the CLASSPATH used to compile the generated files.

This option is the same as the **wsdeploy** command parameter `-jardir` *`directory`*.

Information	Value
Data type	String
Default	null

wsdeploy command
Use the **wsdeploy** command to add WebSphere product-specific deployment classes to a web services-compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file.

The **wsdeploy** command is supported by Java API for XML-based RPC (JAX-RPC) applications. The Java API for XML-Based Web Services (JAX-WS) programming model that is implemented by the application server does not support the **wsdeploy** command. If your web services application contains only JAX-WS endpoints, you do not need to run the **wsdeploy** command, as this command is used to process only JAX-RPC endpoints.

The deployment classes that are added by the **wsdeploy** tool to a web services-compatible EAR file or a JAR file include:
- Stubs
- Serializers and deserializers
- Implementations of service interfaces

This deployment step must be performed at least once, and can be performed more often. Deployment can be performed separately using the **wsdeploy** command, assembly tools, or when the application is installed. When using the **wsadmin** command for installation, specify the **-deployws** option.

The **wsdeploy** command operates as noted in the following list:
- Each module in the enterprise application or JAR file is examined.
- If the module contains web services implementations, indicated by the presence of the `webservices.xml` deployment descriptor, the associated Web Services Description Language (WSDL) files are located and the **WSDL2Java** command is run with the role deploy-server option.
- If the module contains web services clients, indicated by the presence of the client deployment descriptor, the associated WSDL files are located and the **WSDL2Java** command is run with the role deploy-client option.
- The files generated by the **WSDL2Java** command are compiled and repackaged.

See the **WSDL2Java** command for JAX-RPC applications command information to learn more about the files that are generated for deployment.

When the generated files are compiled, they can reference application-specific classes outside the EAR or JAR file, if the EAR or JAR file is not self-contained. In this case, use either the -jardir or -cp option to specify additional JAR or zip files to be added to CLASSPATH variable when the generated files are compiled.

wsdeploy command syntax

The command syntax is noted in the following example:

```
wsdeploy Input_filename Output_filename [options]
```

Required options:
- ***Input_filename***

  Specifies the path to the EAR or JAR file to deploy.
- ***Output_filename***

  Specifies the path of the deployed EAR or JAR file. If *output_filename* already exists, it is silently overwritten. The *output_filename* can be the same as the *input_filename*.

Other options:
- **-jardir** *directory*

  Specifies a directory that contains JAR or zip files. All JAR and zip files in this directory are added to the CLASSPATH used to compile the generated files. This option can be specified zero or more times.
- -**cp** *entries*

  Specifies entries to add to the CLASSPATH when the generated classes are compiled. Multiple entries are separated the same as they are in the CLASSPATH environment variable.
- **-codegen**

  Specifies to generate but not compile deployment code. This option implicitly specifies the -keep option.
- **-debug**

  Includes debugging information when compiling, that is, use javac -g to compile.
- **-help**

  Displays a help message and exit.
- **-ignoreerrors**

  Do not stop deployment if validation or compilation errors are encountered.

- **-keep**

  Do not delete working directories containing generated classes. A message is displayed indicating the name of the working directory that is retained.
- **-novalidate**

  Do not validate the web services deployment descriptors in the input file.
- **-trace**

  Displays processing information, including the names of the generated files.
- **-compliancelevel** *level*

  Sets the JDK level for compiler compliance. Valid values include: 1.4, 5.0, 6.0 (default) and 7.0. This flag is optional.

The following example illustrates how the options are used with the **wsdeploy** command:

```
wsdeploy x.ear x_deployed.ear -trace -keep
Processing web service module x_client.jar.
Keeping directory: f:\temp\Base53383.tmp for module: x_client.jar.
Parsing XML file:f:\temp\Base53383.tmp\WarDeploy.wsdl
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java.
Done processing module x_client.jar.
```

The following messages may be displayed:
- Flag *-f* is not valid.

  Option *f* was not recognized as a valid option.
- Flag *-c* is ambiguous.

  Options can be abbreviated, but the abbreviation must be unique. In this case, the **wsdeploy** command cannot determine which option was intended.
- Flag *-c* is missing parameter *-p*.

  A required parameter for an option is omitted.
- Missing *p* parameter.

  A required option is omitted.

JAX-WS application deployment model

The administration function of the product is enhanced to support installing and deploying Java Application Programming Interface (API) for XML Web Services (JAX-WS) applications like any other WebSphere Application Server applications.

A JAX-WS application is packaged as a web application archive (WAR) file or a WAR module within an Enterprise Archive (EAR) file. The JAX-WS application deployment model is similar to the Java API for XML Remote Protocol Call (JAX-RPC) web services application model. The main differences are JAX-RPC web services application requires you to add additional bindings and deployment descriptors for application deployment. A JAX-WS application does not require additional bindings and deployment descriptors for deployment. You can deploy your JAX-WS applications as you would deploy any other WebSphere Application Server application.

JAX-WS web services is a rewrite of JAX-RPC web services. The table compares the web services stack for both JAX-WS and JAX-RPC web services.

JAX-RPC web services	JAX-WS web services
Bindings are proprietary	Bindings are based on the open source Java API for XML Bindings (JAXB)
Parsing is proprietary	Parsing is based on the open source Java Specification Request (JSR) 173

JAX-RPC web services	JAX-WS web services
No Java annotations support	Support for Java annotations such as `@WebService`, `@WebMethod`, `@WebParam`, `@WebResult`, and `@SOAPBinding`
During deployment, some deployment descriptor files are created in a JAX-RPC based service and client. The following files are created on the services side, when it is an EJB based web service and EJB based module: • `webservices.xml` • <name_of_service>`_mapping.xml` • `ibm-webservices-bnd.xmi` • `ibm-webservices-ext.xmi` When the service is a JavaBeans-based or web module-based service, the following files and deployment descriptors are required: • `webservices.xml` • <name_of_service>`_mapping.xml` • In the `web.xml` file, there is no additional content • `ibm-webservices-bnd.xmi` • `ibm-webservices-ext.xmi` The `web.xml` exists in both EJB and JavaBeans based services. However, there is no additional content added to the file during deployment of a Web service application or module.	For JAX-WS web services, the use of the `webservices.xml` deployment descriptor is optional because you can use annotations to specify all of the information that is contained within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

Starting with WebSphere Application Server Version 7.0 and later, Java EE 5 application modules (web application modules version 2.5 or above, or EJB modules version 3.0 or above) are scanned for annotations to identify JAX-WS services and clients. However, pre-Java EE 5 application modules (web application modules version 2.4 or before, or EJB modules version 2.1 or before) are not scanned for JAX-WS annotations, by default, for performance considerations. In the Version 6.1 Feature Pack for Web Services, the default behavior is to scan pre-Java EE 5 web application modules to identify JAX-WS services and to scan pre-Java EE 5 web application modules and EJB modules for service clients during application installation. Because the default behavior for WebSphere Application Server Version 7.0 and later is to not scan pre-Java EE 5 modules for annotations during application installation or server startup, to preserve backward compatability with the feature pack from previous releases, you must configure either the UseWSFEP61ScanPolicy property in the META-INF/MANIFEST.MF of a web application archive (WAR) file or EJB module or define the Java virtual machine custom property, com.ibm.websphere.webservices.UseWSFEP61ScanPolicy, on servers to request scanning during application installation and server startup. To learn more about annotations scanning, see the JAX-WS annotations information.

Using a third-party JAX-WS web services engine

In certain situations you might need to set up a third-party JAX-WS web services engine. For example, you must set up a third-party JAX-WS web services engine if you need to deploy applications that use a single runtime across various application servers such as WebSphere Application Server, JBoss, and WebLogic, or if you want to build JAX-WS web services applications using third party JAX-WS run-times such as CXF, Axis2, and Metro.

Before you begin

Use of a third-party JAX-WS runtime has limitations. It also requires mandatory configuration changes, and in some cases, it requires manual intervention to resolves issues that occur during deployment and when you run the application. These limitations and issues vary based on the third-party JAX-WS runtime you decide to use. You should understand the limitations for the third-party JAX-WS runtime you are preparing to use before you configure your system to use that implementation.

The following limitations exist regardless of which third-party JAX-WS implementation you use:

- The WebSphere Application Server runtime restricts usage of application modules that use both the JAX-WS implementation provided with WebSphere Application Server, and an external JAX-WS implementation in the same application EAR file. You must use either the JAX-WS implementation provided with WebSphere Application Server or the external implementation in a single application EAR file. This limitation ensures that the runtime WebSphere Application Server does not conflict with the external third-party JAX-WS implementation.
- You must remove any conflicting JAR files from your application library before you deploy an application that uses an external JAX-WS implementation. Most of the external third-party JAX-WS runtimes include some JAR file libraries that are already installed on WebSphere Application Server. This situation causes conflicts in your application library.
- After an application that uses a third-party JAX-WS runtime is deployed on WebSphere Application Server, it is not recognized as a service client or provider. Therefore, you cannot attach application level policy sets to these applications. You must rely on external runtimes support quality of service. Following is a list of WebSphere Application Server features that are not available if you decide to deploy and run application that uses third-party JAX-WS implementations:
  - WS-Security, WS-RM, and WS-Transactions policy sets
  - WSDM
  - JNDI lookup to retrieve JAX-WS Service or Port Instance.

gotcha: Even though IBM supports the enablement of third party JAX-WS runtimes to run on WebSphere Application Server, and ensures the successful deployment of applications that use such runtimes, IBM does not provide support for resolving JAR file conflict problems, or any problem that a stack trace indicates is in the third party code.

About this task

When you deploy an application EAR file with a third-party JAX-WS implementation on WebSphere Application Server, the WebSphere Application Server runtime must ensure the use of the third-party engine, and disable the use of the existing WebSphere Application Server JAX-WS web services engine.

WebSphere Application Server does not claim support for any of the third-party JAX-WS runtimes, but has tested the deployment and execution of applications that use such runtimes.

You must complete the following steps before you can use an external JAX-WS runtime in an application.

Procedure

1. Set the class loader policy to `Classes loaded with local class loader first (parent last)` at the module level.

   Changing the class loader policy to parent last ensures that the external third-party JAX-WS runtime and their dependent library JAR files are first in the class loader search path, thereby ensuring that the third-party implementation is used instead of the WebSphere Application Server.

   a. In the administrative console, click **Applications > Application Types > WebSphere enterprise applications >** *application_name* **> Class loading and update detection**.

   b. Under Class reloading options, select **Override class reloading settings for web and EJB modules** .

c. Under Class loader order, select **Class loader order** property to `Classes loaded with local class loader first (parent last)`.

a. Click **OK**, and then **Save** to save your changes.

2. Turn off web services annotation scanning.

   Annotation scanning can be turned off at the application level or at the server level.

   To turn off annotation scanning at the application level, set the DisableIBMJAXWSEngine property in the META-INF/MANIFEST.MF of a WAR file or EJB module to true. Example:

```
Manifest-Version: 1.0
DisableIBMJAXWSEngine: true
```

   To turn off web services annotation scanning at the server level:

   a. In the administrative console, go to the Custom properties page for the Java virtual machine.

      Click **Servers > Server Types > WebSphere application servers >** *server_name*, and then, in the Server Infrastructure section, click **Java and process management > Process definition > Control > Java virtual machine > Custom properties**

   b. Set the **com.ibm.websphere.webservices.DisableIBMJAXWSEngine** property to `true`

      If this property does not already exist for your configuration, click **New**, and add `com.ibm.websphere.webservices.DisableIBMJAXWSEngine` in the **Name** field and `true` in the **Value** field.

Results

What to do next

- Deploy and run an application EAR file with a third-party JAX-WS implementation on WebSphere Application Server.

Deploying web services client applications

After you have created an enterprise archive (EAR) file for the web services client application, you can deploy the web services client application into the Application Server.

Before you begin

To deploy a Java-based web services client, you need an enterprise application, also known as an enterprise archive (EAR) file that is configured and enabled for web services.

A Java API for XML-Based Web Services (JAX-WS) application is packaged as a web application archive (WAR) file or a WAR module within an Enterprise Archive (EAR) file. A JAX-WS application does not require additional bindings and deployment descriptors for deployment whereas a Java API for XML-based RPC (JAX-RPC) web services application requires you to add additional bindings and deployment descriptors for application deployment. JAX-WS is much more dynamic, and does not require any of the static data generated by the deployment step required for deploying JAX-RPC applications. For JAX-RPC web services clients, you must configure the client deployment descriptors.

About this task

You can use either the administrative console or the **wsadmin** scripting tool to deploy an EAR file. If you are installing an application containing web services by using the **wsadmin** command, specify the `-deployws` option for JAX-RPC applications.

Use the **wsdeploy** command only with JAX-RPC applications. The **wsdeploy** command is not applicable for JAX-WS applications.

If you are installing an application containing web services by using the administrative console, select **Deploy WebServices** in the Install New Application wizard. Read about installing a new application for more information on using the administrative console.

The following actions deploy the EAR file with the **wsadmin** command:

Procedure

1. Start *install_root*/bin/wsadmin from a command prompt.
2. Deploy the EAR file.
   - For JAX-WS web service applications, enter the `$AdminApp install` *EARfile* `"-usedefaultbindings"` command at the **wsadmin** prompt.
   - For JAX-RPC web service applications, enter the `$AdminApp install` *EARfile* `"-usedefaultbindings -deployws"` command at the **wsadmin** prompt.

Results

You have a deployed a web service client in the application server runtime environment.

What to do next

Test the web services client. You can now test a web services-enabled managed client EAR file or an unmanaged client JAR file.

Making deployed web services applications available to clients

You can publish WSDL files to the file system. If you are a client developer or a system administrator, you can use WSDL files to enable clients to connect to web services.

Before you begin

The publish WSDL administrative console panel supports both JAX-RPC and JAX-WS services. The publish WSDL panel generates a compression file that contains WSDL files for all modules in an application that contains JAX-WS or JAX-RPC web services. Read about providing the HTTP endpoint URL information to learn how the URL information affects the content of the published WSDL.

To publish a Web Services Description Language (WSDL) file you need an enterprise application, also known as an enterprise archive (EAR) file, that contains a Web services-enabled module and has been deployed into WebSphere Application Server. To learn how to deploy web services, see the deploying web services applications onto application servers information.

About this task

The purpose of publishing the WSDL file is to provide clients with a description of the web service, including the URL identifying the location of the service.

After installing a web services application, and optionally modifying the endpoint information, you might need WSDL files containing the updated endpoint informations to make deployed web services application to be available to clients.

Before you publish a WSDL file, you can configure web services to specify endpoint information in the form of URL fragments to enable full URL specification of WSDL ports. Refer to the tasks describing configuring endpoint URL information.

The WSDL files for each web services-enabled module are published to the file system location you specify. You can provide these WSDL files to clients that want to invoke your Web services.

You can specify endpoint information for HTTP ports, for Java Message Service (JMS) ports, or you can directly access enterprise beans that are acting as web services.

Procedure

1. Configure the web services client bindings.
2. Configure the URL endpoint information for HTTP bindings. Do one of the following depending on what kind of bindings you are using:
   - Configure the URL endpoint information for HTTP bindings.
   - Configure the URL endpoint information for JMS bindings.
   - Configure the URL endpoint information to directly access enterprise beans.
3. Externalize or publish the WSDL file out of the application. You can complete this task in the following ways:
   - Publish a WSDL file with the administrative console
   - Publish a WSDL file using a URL.
   - Publish a WSDL file with the wsadmin tool.

What to do next

Apply security to your web services. To learn more, see the securing web services applications using message level security information.

Configuring web services client bindings

When a web services application is deployed into WebSphere Application Server, an instance is created for each application or module. The instance contains deployment information for the web module or Enterprise JavaBeans (EJB) module, including client bindings.

Before you begin

Deploy a web service into your WebSphere Application Server instance. Read about deploying web services applications onto application servers.

You must know the topology of the URL endpoint address of the web services servers and which web service the client depends upon. You can view the deployment descriptors in the administrative console to find the topology information. To learn more, see the View web services server deployment descriptors information.

About this task

The client bindings define the Web Services Description Language (WSDL) file name and preferred ports. The relative path of a web service in a module is specified within a compatible WSDL file that contains the actual URL to be used for requests. The address is only needed if the original WSDL file did not contain a URL, or when a different address is needed. For a service endpoint with multiple ports, you need to define an alternative WSDL file name.

The following steps describe how to edit bindings for a web service after these bindings are deployed on a server. When one web service communicates with another web service, you must configure the client bindings to access the downstream web service.

You can also configure client bindings with the wsadmin tool. Read about configuring a web service client deployed WSDL file name with the wsadmin tool.

To configure client bindings through the administrative console:

Procedure

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > *application_instance* > **Manage Modules** > *module_instance* > **Web services client bindings**.
3. Find the web service you want to update.

   The web services are listed in the **Web Service** field.
4. Select the WSDL file name from the drop down box in the WSDL file name field.
5. Click **Edit** in the Preferred port mappings field to configure the default port to use.
   a. Specify the port type and the preferred ports in the Port type and Preferred ports fields.

      Configuring the preferred port enables you to select an optimal port implementation use non-SOAP protocols. See the RMI-IIOP web services using JAX-RPC information to learn more about using non-SOAP protocols.
   b. Click **Apply** and **OK**.
6. Click **Edit** in the Port information field to configure the request timeout, the overridden endpoint, and the overridden binding namespace for a port.

   Configuring the request timeout accommodates complex topologies that can have multiple cascaded Web services that involve multiple hops or long-running services.

   You can configure Timeout values based on observed behavior of the overall system as integration proceeds. For example, a web service client might time out because of changing network conditions or the performance of an external web service. When you have applications containing web services clients that timeout, you can change the request time out values for the clients.

   You can specify an endpoint URL to override the current endpoint. A client invoking a request on this port uses this endpoint instead of the endpoint specified in the WSDL file. You can specify the **Overridden endpoint URL** value for both Java API for XML-Based Web Services (JAX-WS) clients and Java API for XML-based RPC (JAX-RPC) clients.

   Note: The **Overridden endpoint URL** field is applicable for both JAX-WS and JAX-RPC clients. The other fields on this administrative console page are only applicable for JAX-RPC clients.
   a. Click **Apply** and **OK**.

Results

Your web service client bindings are configured.

What to do next

Now you can finish any other configurations, start or restart the application, and verify the expected behavior of the web service.

Web services client bindings

The client bindings define the Web Services Description Language (WSDL) file name, preferred ports and other port information. Use this page to specify the client bindings and the port mappings for the web services in a module.

A web service can specify the relative path within the module of a compatible WSDL file containing the actual URL to be used for requests. The relative path only needs to be specified if the original WSDL file does not contain a URL or when a different URL is needed. For a service endpoint with multiple ports defined, a preferred mapping specifies the default port to use for a port type.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***resource_name*** .
2. Click **Manage Modules** > ***module_name*** > **Web services client bindings**.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

Web service:

Identifies the name of this web service. A module can contain one or more web services.

EJB:

For EJB modules, identifies the name of the EJB.

WSDL file name:

Specifies the WSDL file name, which is relative to the module. Locate the WSDL file name in the drop down menu.

Preferred port mappings:

Specifies and manages the preferred port type mapping for a web service when a particular port type is requested.

Click **Edit** to edit the preferred port mapping information on the Preferred port mappings page.

Port information:

Specifies additional configuration information for the ports of this web service.

Click **Edit** to edit the port information on the Port information page. You can set a request timeout, override an endpoint and override a binding namespace for each client port.

Preferred port mappings:

Use this page to view and manage a preferred portType mapping for a web service.

This administrative console page applies only to Java API for XML-based RPC (JAX-RPC) applications.

When you have multiple ports that reference the same portType (service endpoint interface), a preferred port specifies the port to use when the `Service.getPort(Class SEI)` method is called with only the service endpoint interface.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Manage Modules** > ***module instance*** > **Web services client bindings** > **Edit** > ***preferred_port_instance***.

portType:

Specifies the portType.

The preferred port and the portType values are both of the type `java.xml.namespace.QName`.

Preferred port:

Specifies the preferred port to be associated with a particular portType. The `Service.getPort(Class)` method returns the preferred port associated with the specified service endpoint interface class (portType).

The preferred ports available are listed, as well as a value of `None`, which indicates no preferred port is selected.

Web services client port information:

Use this page to specify a request timeout, override an endpoint, and override a binding namespace for a web services client port.

A web service can have multiple ports. You can view and configure the port attributes for each defined web service port. The web services are listed on the web services client bindings page.

To view this administrative console page, complete the following steps:

1. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***resource_name*** .
2. Click **Manage Modules** > ***module_name*** > **Web services client bindings**.
3. Click **Edit** under **Port Information**.

This administrative console page applies to both Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services. The **Overridden endpoint URL** field is the only field supported for JAX-WS clients. The other fields are not applicable for JAX-WS clients.

Port:

Specifies the name of a port.

Request timeout:

Specifies the time, in seconds, that a web service client waits for a request to complete on this port. If a timeout is not specified, the default request timeout for the client to wait is `300` seconds. If the value is set at `0` (zero), the timeout used is the default value for the underlying transport mechanism. This field is supported only for JAX-RPC clients.

A typical use for this setting is to customize the client's behavior when it is configured to use a JMS transport to access a web service to make it wait longer for an expected completion. Depending upon network conditions, or the nature of a web service implementation, it might be necessary to tune the timeout.

Overridden endpoint URL:

Specifies the name of an endpoint that is used to override the current endpoint. A client invoking a request on this port uses this endpoint instead of the endpoint specified in the WSDL file. This field is supported for both JAX-WS and JAX-RPC clients.

If an assembled application contains a web service client that is statically bound, the client is locked into using the implementation (service end point) identified in the WSDL file used during development. Overriding the endpoint is an alternative to configuring the deployed WSDL attribute.

The overridden endpoint URI attribute is specified on a per port basis. It does not require an alternative WSDL file within the module. The overridden endpoint URI takes precedence over the deployed WSDL attribute. The client uses this value for the service end point URI or SOAP address, instead of the value in the static client bindings.

Note: You can edit this field if you have managed clients or a mixture of both managed and unmanaged clients. You cannot edit the field if you have unmanaged clients only.

If you do not want a request by an unmanaged JAX-WS client service to be sent to the endpoint URL that is specified in this field, you can specify the

com.ibm.ws.websvcs.unmanaged.client.dontUseOverriddenEndpointUri Java virtual machine (JVM) system property. For more information about this custom property, read about the Java virtual machine custom properties.

Overridden binding:

Specifies the WSDL file binding namespace URI to use with this port, instead of the namespace in the WSDL file. This binding does not need to exist in the WSDL file. A client invoking a request on this port uses this binding instead of the binding specified in the WSDL file. An overridden binding namespace cannot be specified unless an overridden endpoint is specified. This field is supported only for JAX-RPC clients.

Configuring endpoint URL information for HTTP bindings

Configuring a service endpoint is necessary to connect Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services clients to any web services among the components being assembled or to any external web services.

Before you begin

You can develop an HTTP accessible Java API for XML-based remote procedure call (JAX-RPC) or Java API for XML Web Services (JAX-WS) web service when you have an existing JavaBeans object to enable as a web service. For additional information, see the using HTTP to transport web services requests information.

You can use either the administrative console or property files to configure and manage HTTP endpoint URL fragments. To learn about using property files to set and manage the URL fragments, see the information about working with web services endpoint URL fragment property files.

This task describes using the administrative console to configure endpoint URL information for HTTP bindings.

About this task

You can specify HTTP URL prefixes for web services that are accessed through HTTP by using the Provide HTTP endpoint URL information panel in the administrative console. The HTTP URL prefixes provide location specific information and are used to form complete endpoint URLs that are included within published WSDL files.

Note: The Provide HTTP panel in the administrative console displays modules that contain Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services. You can use the Provide HTTP panel to provide URL information for both types of web services, however, the panel does not indicate which type of service that you are working with.

To configure these prefixes with the administrative console:

Procedure

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > *application_instance* > **Provide HTTP endpoint URL information**.
3. Specify the URL prefixes for the web service.

   In this step you specify the protocol (HTTP or HTTPS), as well as the *host_name* and *port_number* used in the endpoint URL. You can select a prefix from a predefined list, by selecting the default HTTP URL prefix, or you can use a custom HTTP URL prefix.

   a. Select **Default HTTP URL prefix** or **Custom HTTP URL Prefix**.

If you select the default HTTP URL prefix, a list provides you with a choice of endpoint URL prefixes. The list is a combination of two sets of ports in the module: the virtual host ports and the application server ports. Use a prefix from this list if the application server of the web service is accessed directly. Select a value and also select the check box of the modules to use the prefix.

If you want to use a custom HTTP URL prefix, type the value in the field. Select the check box to use in the prefix.

If you configure a custom HTTP URL prefix, , you must also configure the custom JVM property, `com.ibm.ws.webservices.enableHTTPPrefix` in the administrative console and set the value to `true`. You must restart the application server after this custom property has been defined so that this property is used by the system. Setting this custom JVM property is required so the custom HTTP endpoint prefix information is correctly displayed in the ?WSDL query that is returned from the browser and the URL field of the WSDL file that is returned to the client. If this custom property is not defined with the value of `true`, the custom HTTP URL prefix is not reflected in the WSDL file that the service returns to the client. To learn how to configure this custom JVM property, see the documentation on configuring additional HTTP transport properties using the JVM custom property panel in the administrative console.

Note: The `com.ibm.ws.webservices.enableHTTPPrefix` custom property applies to JAX-RPC web services applications only.

b. Click **Apply**.

The URL prefix, whether default or custom, is copied to the selected module HTTP URL prefix field.

c. Click **OK**. The URL information is saved to your workspace.

Results

You have specified the partial URL information that is used to form the target endpoint addresses in the WSDL files that are published using the Publish WSDL files panel.

What to do next

Configure any other URL endpoint information for Java Message Service (JMS) bindings and direct Enterprise JavaBeans (EJB) access. Then publish the WSDL files to make the deployed web services application available to clients.

Provide HTTP endpoint URL information

Use this page to specify endpoint URL prefix information for web services accessed by HTTP. Prefixes are used to form complete endpoint addresses included in published Web Services Description Language (WSDL) files.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Provide HTTP endpoint URL information**.

You can specify a portion of the endpoint URL to be used in each web service module. In a published WSDL file, the URL defining the target endpoint address is found in the location attribute of the port's `soap:address` element.

This administrative console page applies for Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

In addition to the using the administrative console, you can use property files to configure and manage HTTP endpoint URL fragments. To learn about using property files to set and manage the URL fragments, see the information about working with web services endpoint URL fragment property files.

Specify endpoint URL prefixes for web services:

Specifies the *protocol* (either http or https), *host_name*, and *port_number* to be used in the endpoint URL.

You can select a prefix from a predefined list using the **HTTP URL prefix** or **Custom HTTP URL prefix** field.

The URL prefix format is *protocol://host_name:port_number*, for example, `http://myHost:9045`. The actual endpoint URL that is contained in a published WSDL file consists of the prefix followed by the module's context-root and the web service url-pattern, for example, `http://myHost:9045/services/myService`.

Select default HTTP URL prefix:

Specifies the drop down list associated with a default list of URL prefixes. This list is the intersection of the set of ports for the module's virtual host and the set of ports for the module's application server. Use items from this list if the web services application server is accessed directly.

To set an HTTP endpoint URL prefix, select **Select default HTTP URL prefix** and select a value from the drop down list. Select the check box of the modules that are to use the prefix and click **Apply**. When you click **Apply**, the entry in the **Select default HTTP URL prefix** or **Select custom HTTP URL prefix** fields, depending on which is selected, is copied into the **HTTP URL prefix** field of any module whose check box is selected.

Select custom HTTP URL prefix:

Specifies the *protocol*, *host*, and *port_number* of the intermediate service if the web services in a module are accessed through an intermediate node, for example the web services gateway or an IHS server.

To set a custom HTTP endpoint URL prefix, you must also configure the custom JVM property, `com.ibm.ws.webservices.enableHTTPPrefix` in the administrative console and set the value to `true`. Setting this custom JVM property is required so the custom HTTP URL is correctly populated in the URL field of the WSDL file that is returned to the client. If this custom JVM property is not configured, the custom HTTP URL prefix is not in the URL field in the copy of the WSDL file that the service returns to the client. To learn how to configure this custom JVM property, see the documentation on configuring additional HTTP transport properties using the JVM custom property panel in the administrative console. You must restart the application server after this custom property has been defined so that this property is used by the system.

After the `com.ibm.ws.webservices.enableHTTPPrefix` custom JVM property is configured, select **Select custom HTTP URL prefix** and enter a value. Select the check box of the modules that are to use the prefix and click **Apply**. When you click **Apply**, the entry in the **Select default HTTP URL prefix** or **Select custom HTTP URL prefix** fields, depending on which is selected, is copied into the **HTTP endpoint URL prefix** field of any module whose check box is selected.

Note: The `com.ibm.ws.webservices.enableHTTPPrefix` custom property applies to JAX-RPC web services applications only.

Configuring endpoint URL information for JMS bindings

WebSphere Application Server supports the use of the Java Message Service (JMS) API to transport web services requests, as an alternative to using HTTP.

Before you begin

The application server supports use of the Java Message Service (JMS) API to transport web services requests, as an alternative to HTTP transport. Read about using the Java Message Service (JMS) to transport web services requests to learn more about how web service clients and servers can communicate through JMS queues and topics instead of through HTTP connections.

You can use either the administrative console or property files to configure and manage JMS endpoint URL fragments. To learn about using property files to set and manage the URL fragments, see the information about working with web services endpoint URL fragment property files.

This task describes using the administrative console to configure endpoint URL information for JMS bindings.

About this task

Configuring a service endpoint is necessary to connect web service clients to any web services among the components being assembled or to any external web services. You can configure the endpoint URL information for JMS during application installation

In this task, enter the JMS endpoint URL prefix to use for each web service-enabled Enterprise JavaBeans (EJB) Java archive (JAR) file that belong to the application. The JMS endpoint URLs are included in the Web Services Description Language (WSDL) files published for clients to use.

You can specify HTTP URL prefixes for web services that are accessed through HTTP by using the Provide HTTP endpoint URL information panel in the administrative console. These prefixes are used to form complete endpoint addresses that are included in WSDL files when published.

You can specify JMS URL prefixes by using the Provide JMS and EJB endpoint URL information panel in the administrative console during or after application installation.

This task applies for Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

To configure JMS URL prefixes:

Procedure

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > ***application_instance*** > **Provide JMS and EJB endpoint URL information**.
3. Locate the list of web services modules that are accessible through JMS transport.
4. Type the JMS URL fragment in the **URL fragment** field. Enter a URL fragment that is a prefix to the initial URL part that is obtained by examining the deployment information of the Web service. See the usage scenario following this task for more information.

   The value that you enter is used to define the location attribute of the port soap:address element within the WSDL file that is published using the *application_name*_ExtendedWSDLFiles.zip or the *application_name*_WSDLFiles.zip file on the Publish WSDL zip files panel.

Results

You have a web service that is accessible through the JMS transport and configured with JMS bindings.

Example

Suppose an application called StockQuoteService contains an EJB JAR file that is named StockQuoteEJB, which contains one or more web services that are accessible through the JMS transport.

See the using SOAP over Java Message Service to transport web services information to review the example that defines a queue with the Java Naming and Directory Interface (JNDI) name of jms/StockQuote_Q, and a connection factory with the JNDI name of jms/StockQuote_CF, for your application.

In this example, you specify the following string as the JMS URL prefix within the Provide JMS and EJB endpoint URL information panel:

```
jms:/queue?destination=jms/StockQuote_Q&connectionFactory=jms/StockQuote_CF
```

The WSDL publisher uses this partial URL string to produce the actual JMS URL for each port component that is defined in the module. The `targetService=<port_name>` string is added to the end of the JMS URL, for example:

```
jms:/queue?destination=jms/StockQuote_Q&connectionFactory=jms/StockQuote_CF&targetService=getQuote
```

The published WSDL file is used by clients to invoke the web service.

What to do next

Publish the WSDL files to make the deployed web services application available to clients.

Provide JMS and EJB endpoint URL information

Use this page to specify Java Message Service (JMS) and Enterprise JavaBeans (EJB) endpoint URL fragments for web services accessed through SOAP and Java Message Service (JMS) or directly as enterprise beans. Fragments are used to form complete endpoint addresses included in published Web Services Description Language (WSDL) files.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Provide JMS and EJB endpoint URL information**.

You can specify a fragment of the endpoint URL to be used in each web service module. In a published WSDL file, the URL defining the target endpoint address is found in the location attribute of the port's `soap:address` element.

If you are using web services modules that are configured to use JMS or configured to access enterprise beans directly, these modules are listed on this panel.

This administrative console page applies for Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

In addition to the using the administrative console, you can use property files to configure and manage JMS and EJB endpoint URL fragments. To learn about using property files to set and manage the URL fragments, see the information about working with web services endpoint URL fragment property files.

URL fragment for JMS:

Specifies a URL fragment for web services accessed through a JMS transport. You can enter a value that is used to define the `soap:address` of a web service. When WSDL files are published, a URL is formed using this fragment and is contained in the WSDL files.

The URL fragment that is entered as a value is a prefix to which the targetService property is appended to form a complete JMS URL endpoint. The default value is obtained by examining the installed service's deployment information, for example, `jms:jndi:jms/MyQueue&jndiConnectionFactoryName=jms/MyCF`.

This information is obtained from the configured JMS endpoint for the web service, which is a Message Driven Bean (MDB) defined by the **endpointEnabler** command-line tool. You can modify the URL fragment, for example, by adding properties. The URL fragment is combined with the targetService property to form the complete URL, for example, `jms:jndi:jms/MyQueue&jndiConnectionFactoryName=jms/MyCF&priority=5&targetService=GetQuote`.

URL fragment for EJB:

Specifies a URL fragment for web services accessed through an EJB binding. You can enter a value used to define the location attribute of the port's `generic:address` element of a Web service. This port address is contained in the WSDL compression file when the compression file is published using the *application_name_ExtendedWSDLFiles*`.zip` field on the **Publish WSDL zip file** panel.

The URL fragment value entered is a suffix, which is appended to the initial part of the URL obtained by examining the web service's deployment information. For example, the following URL fragment can be obtained from the EJB's deployment information: `wsejb:/com.acme.sample.MyStockQuoteHome?jndiName=ejb/MyStockQuoteHome`.

In this case, you can enter the following information in the URL fragment field, `jndiProviderURL=corbaloc:iiop:myhost.mycompany.com:2809`, which results in this endpoint URL, `wsejb:/com.acme.sample.MyStockQuoteHome?jndiName=ejb/MyStockQuoteHome &jndiProviderURL=corbaloc:iiop:myhost.mycompany.com:2809`.

Configuring endpoint URL information to directly access enterprise beans

WebSphere Application Server supports directly accessing an enterprise bean as a web service, as an alternative to using HTTP or Java Message Service (JMS) to transport requests between the server and the client. The Enterprise JavaBeans (EJB) module that is used as a web service contains a Web Services Description Language (WSDL) file that contains EJB bindings.

Before you begin

To learn more about the process of directly accessing an enterprise bean as a web service, see the using WSDL EJB bindings to invoke an EJB from a JAX-RPC web services client.

You can use either the administrative console or property files to configure and manage EJB endpoint URL fragments. To learn about using property files to set and manage the URL fragments, see the information about working with web services endpoint URL fragment property files.

This task describes using the administrative console to configure endpoint URL information to directly access enterprise beans.

About this task

Configuring a service endpoint is necessary to connect web service clients to any web services among the components being assembled or to any external web services.

You can specify web address endpoints of the enterprise bean for web services that are accessed directly by EJB bindings using the Provide JMS and EJB endpoint web address information panel in the administrative console.

If you have modules that are configured for using direct EJB access, the modules are listed on the Provide JMS and EJB endpoint web address information panel in the administrative console. The EJB endpoint is only available in the WSDL that is found in the *application_name*`_ExtendedWSDLfiles.zip` file.

You can specify a fragment of the endpoint web address for the web services in each module.

To configure the web address endpoints of the enterprise bean with the administrative console:

Procedure

1. Open the administrative console.
2. Click **Applications** > **Enterprise Applications** > ***application_instance*** > **Provide JMS and EJB endpoint URL information**.

3. Locate the list of EJB modules.
4. Select the application module.
5. Type the web address fragment in the **URL fragment** field.

   Enter a web address fragment that is a suffix to the initial web address part that is obtained by examining the web service deployment information. See the example following this task for more information.

   The value that you enter is used to define the location attribute of the port generic:address element within the WSDL file that is published using the *application_name*_ExtendedWSDLFiles.zip file name link on the Publish WSDL zip files panel. The zip file names are listed as links on the panel.
6. Click **OK**.
7. Click **Save**.

Results

You have configured endpoints of the enterprise bean for Web services that are accessed directly by EJB bindings.

Example

The following example illustrates a web address fragment to enter in the URL fragment field.

The following web address information can be obtained from the deployment descriptor of an enterprise bean:

```
wsejb:/com.acme.sample.MyStockQuoteHome?jndiName=ejb/MyStockQuoteHome
```

Enter the following web address fragment in the URL fragment field:

```
jndiProviderURL=corbaloc:iiop:myhost.mycompany.com:2089
```

The results are shown in the following example:

```
wsejb:/com.acme.sample.MyStockQuoteHome?jndiName=ejb/MyStockQuoteHome&jndiProviderURL=
corbaloc:iiop:myhost.mycompany.com:2089
```

What to do next

Provide a description of the web service to the service requestor by publishing WSDL files. To learn more, read about making deployed web services applications available to clients.

Publishing WSDL files using the administrative console

You can publish a Web Services Description Language (WSDL) file using the WebSphere Application Server administrative console.

Before you begin

Before completing this task, you need to install or deploy the web service. After deployment, configure the URL endpoint tasks for your transport:

- Configure endpoint URL information for HTTP bindings
- Configure endpoint URL information for JMS bindings
- Configure endpoint URL information to directly access enterprise beans

About this task

By publishing a WSDL file, you are providing clients with a description of the web service, including the URL identifying the location of the service.

The WSDL files in each web services-enabled module are published to the file system location you specify. You can provide these WSDL files in the development and configuration process of the web service clients so they can invoke your web services.

This task applies for Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

To learn about more ways to publish WSDL files, see the making deployed web services applications available to clients information.

To publish an application's WSDL file with the administrative console:

Procedure

1. Open the administrative console.
2. Click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name***.
3. Under Web Services Properties, click **Publish WSDL files**. This takes you to the **Publish WSDL zip files** page.
4. Click the WSDL compression file to download. The compression file contains the application's published WSDL files. The compression file `ExtendedWDLFiles.zip` contains EJB binding information. It can also contain JMS or HTTP binding information. The compression file `WSDLFiles.zip` only contains JMS or HTTP binding information.

   The compression file `ExtendedWDLFiles.zip` can contain HTTP binding information. The compression file `WSDLFiles.zip` only contains HTTP binding information.

What to do next

Apply security to your web services. To learn more, see the securing web services applications using message level security information.

Publish WSDL compressed files settings

Use this page to publish Web Services Description Language (WSDL) files.

This administrative console page applies for Java API for XML-Based Web Services (JAX-WS) and Java API for XML-based RPC (JAX-RPC) web services.

The publish WSDL panel generates a compressed file that contains WSDL files for all modules in an application that contains a JAX-WS or JAX-RPC web service. Read about providing the HTTP endpoint URL information to learn how the URL information affects the content of the published WSDL.

To view this administrative console page, click **Applications** > **Application Types** > **WebSphere enterprise applications** > ***application_name*** > **Publish WSDL files**.

When you click **OK**, a panel showing one or several compressed file names displays. Each compressed file contains a WSDL file that represents the web services-enabled modules in the application. When you select a compressed file to publish, a dialogue displays from which you can choose where to create the compressed file. Within the published compressed files, the directory structure is *application_name*/*module_name*/[META-INF|WEB-INF]/wsdl/*wsdl_file_name*.

In a published WSDL file, the location attribute of a port's `soap:address` element contains the endpoint URL through which the web service is accessed. Using the **Provide HTTP endpoint URL information** and the **Provide JMS and EJB endpoint URL information** panels, configure the endpoint URLs to be used for the web services in each module.

*application_name*_***WSDLFiles.zip:***

Specifies the *application_name*_WSDLFiles.zip file containing the WSDL that describes web services that are accessible by standard SOAP-based ports.

*application_name*_***ExtendedWSDLFiles.zip:***

Specifies the *application_name*_ExtendedWSDLFiles.zip file containing the WSDL file that describes the web services available, including SOAP-based and non-SOAP based (for example, EJB) ports.

If there are no web services configured for direct EJB access, this compressed file name is not displayed. Do not use this compressed file if you want to produce a WSDL file compliant to standards.

Publishing WSDL files using a URL

You can publish a Web Services Description Language (WSDL) file using a URL.

Before you begin

Before you can publish a WSDL file using a URL, ensure the web services-enabled application is installed and running.

The files referenced by the <wsdl-file> element in the webservices.xml might import other WSDL or XML Schema Definition (XSD) files. Typically, all WSDL or XSD files are initially placed into the META-INF/wsdl directory when using Enterprise JavaBeans (EJB) or the WEB-INF/wsdl directory when using JavaBeans. If your WSDL or XSD files are not placed in one of these directories, the file referenced by the <wsdl-file> and its imported files are copied to the wsdl directory for publishing purposes.

There are two different forms of URL query strings. The first appends /wsdl to the service and returns only HTTP and JMS bindings. The second appends /extwsdl to the service and returns the extended WSDL file, including HTTP, JMS, and EJB bindings. If a WSDL file contains only EJB bindings and the /wsdl query is used, an error message displays in the browser saying there are no HTTP or JMS bindings in the WSDL file. The error message suggests using the /extwsdl query instead. Publishing a WSDL file using a URL requires that the application have a web module; either provided by the application or in the form of an HTTP router module. If an EJB application contains a WSDL file with only JMS or EJB Web service bindings, the **endptEnabler** command can be used to add an HTTP router module to the application.

Note: Only HTTP URLs are supported for publishing.

About this task

To publish a WSDL file using a URL:

Procedure

1. Retrieve the outer-most WSDL file. The outer-most WSDL file is the WSDL file defined by the <wsdl-file> element in the webservices.xml file.

   Each web service has an endpoint address, like http://example.com/services/stockquote. You can retrieve the outer-most WSDL file (defined by the <wsdl-file> element within the webservices.xml file) by appending the string "/wsdl" or "/wsdl/" to the endpoint address, for example, http://example.com/services/stockquote/wsdl.
2. Retrieve the imported WSDL files. When the outer-most WSDL file imports other WSDL or XSD files, these imported files can be retrieved by appending the relative path to the URL, which is used to retrieve the outer-most WSDL file. This is also true for WSDL files that import other files. This process is similar to the use of relative hyperlinks in HTML documents. If an HTML document contains a hyperlink to other documents, the relative path is appended to create the URL to access the hyperlinked documents.

Example

Suppose you have an application with the following directory structure:

```
<module-root>/
 WEB-INF/
  webservices.xml    /* the <wsdl-file> element points to "WEB-INF/wsdl/fooImpl.wsdl"*/
  web.xml
  ibm-webservices-bnd.xml

  wsdl/
   fooImpl.wsdl   /* imports foo.wsdl which is an interface wsdl */
   foo.wsdl       /* type definition for the interface */
```

If the SOAP address for the foo service is `http://examples.com:9080/services/foo`, the simple way to retrieve the foo service's outer-most WSDL is with the following form: `http://examples.com:9090/services/foo/wsdl` or `http://examples.com:9090/services/foo/wsdl/`. The URL is redirected to `http://examples.com:9090/services/foo/wsdl/fooImpl.wsdl`, where `fooImpl.wsdl` is the name of the outer-most WSDL file.

Since the `fooImpl.wsdl` file has the import `<import namespace="http://examples.com/foo" location="a/b/foo.wsdl>`, use the URL `http://examples.com:9090/services/foo/wsdl/a/b/foo.wsdl` to obtain the `foo.wsdl` file.

Running an unmanaged web services JAX-RPC client

WebSphere Application Server Version 8.5 and the Application Client for WebSphere Application Server Version 8.5 provides a thin Java Platform, Standard Edition 6 (Java SE 6) web services client runtime implementation that is based on the Java API for XML-based RPC (JAX-RPC) 1.1 specification. The Thin Client for JAX-RPC with WebSphere Application Server is a stand-alone Java SE 6 client environment that enables running unmanaged JAX-RPC web services client applications in a non-WebSphere environment to invoke web services that are hosted by the application server.

Before you begin

Note: You can use the Thin Client for JAX-RPC with WebSphere Application Server as a stand-alone client run time in a pure Java SE environment, or within an OSGi environment. The Thin Client for JAX-RPC is not supported when running within WebSphere Application Server or WebSphere Application Client environments. In this version of the application server, with the exception of the Administration Thin Client, other Thin Client run times provided with the application server can also reside in the CLASSPATH and coexist with the Thin Client for JAX-RPC.

Before you can set up a JAX-RPC unmanaged client environment you will need to obtain the Thin Client for JAX-RPC Java archive (JAR) file. To obtain the Thin Client for JAX-RPC, you must either install the application server or the application client.

The Thin Client for JAX-RPC JAR file, `com.ibm.ws.webservices.thinclient_8.5.0.jar`, is located in the *app_server_root*`\runtimes` directory. Refer to the license agreements to ensure correct usage and for limitations on copies of the Thin Client for JAX-RPC outside of the WebSphere environment.

The Thin Client for JAX-RPC is supported in the following environments:

- IBM Software Development Kits (SDKs) Version 6.0
- Sun Java Development Kit (JDK) Version 6.0 that are provided by IBM
- non-IBM SDKs Version 6.0 with this limitation:
  - Xerces limitation on non-IBM SDKs

    If you are using a non-IBM SDK, because of dependencies with the Xerces implementation, you will need to download Xerces-J version 2.6.2 and set it in the classpath before attempting to run the Thin Client for JAX-RPC.

– Equinox 3.6 OSGi runtime environments

About this task

Note: WS-Addressing is not supported for JAX-RPC web services in an unmanaged client environment. If you need to use WS-Addressing, or a web service standard that relies on WS-Addressing, such as WS-Notification, you must use the Thin Client for Java API for XML-based Web Services (JAX-WS) instead. To learn how to setup and run the Thin Client for JAX-WS, see the Thin Client for JAX-WS documentation.

Procedure

1. Configure the path. You can add the Java `bin` directories to your path by typing:

```
set PATH=<your_JDK_bin_directory>;%PATH%
```

2. Configure the classpath.

```
set CLASSPATH=.;<your_web_services_thin_client_install_directory>\com.ibm.ws.webservices.thinclient_8.5.0.jar;
<your_application_jars>;%CLASSPATH%
```

   - If you are using a non-IBM SDK, obtain a Xerces `xml-apis.jar` and `xercesImpl.jar` from the Xerces web site and configure the classpath definition.

```
set CLASSPATH=.;<your_Xerces_install_directory>\xml-apis.jar;<your_Xerces_install_directory>
\xercesImpl.jar;%CLASSPATH%
```

3. Configure SSL for the client.
   a. Add the following system properties to the Java command:

```
-Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props
```

      You can obtain the `ssl.client.props` file from the WebSphere Application Server installation and modify the file to suit your environment. You must, at a minimum, update the location of the `com.ibm.ssl.keyStore` and `com.ibm.ssl.trustStore` key files in the `ssl.client.props` file to the match location of your target environment. For example, use these SSL configuration settings when running the application with a Sun JRE:

```
com.ibm.ssl.protocol=SSL
com.ibm.ssl.trustManager=SunX509
com.ibm.ssl.keyManager=SunX509
com.ibm.ssl.contextProvider=SunJSSE

com.ibm.ssl.keyStoreType=JKS
com.ibm.ssl.keyStoreProvider=SUN
com.ibm.ssl.keyStore=/home/user1/etc/key.jks

com.ibm.ssl.trustStoreType=JKS
com.ibm.ssl.trustStoreProvider=SUN
com.ibm.ssl.trustStore=/home/user1/etc/trust.jks
```

      The key store file and trust store file must be created using the Java keytool utility before the application runs. The automatic key file generation is not supported with a non-IBM product JRE.

4. Enter the following command to run your client application:

```
%JAVA_HOME%/bin/java -Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props <your_client_application>
```

Results

You have set up an unmanaged JAX-RPC client runtime environment that can be used to invoke web services hosted on a WebSphere Application Server.

Running an unmanaged web services JAX-WS client

WebSphere Application Server provides a thin Java Platform, Standard Edition 6 (Java SE 6) web services client runtime implementation that is based on the Java API for XML-based Web Services (JAX-WS) 2.2 specification. The Thin Client for JAX-WS with WebSphere Application Server is a stand-alone Java SE 6 client environment that enables running unmanaged JAX-WS web services client applications in a non-WebSphere environment to invoke web services that are hosted by the application server.

Before you begin

Note: You can use the Thin Client for JAX-WS with WebSphere Application Server as a stand-alone client run time in a pure Java SE environment, or within an OSGi environment. The Thin Client for JAX-WS is not supported running within WebSphere Application Server or WebSphere Application Client environments. In this version of the application server, with the exception of the Administration Thin Client, other Thin Client run times provided with the application server can also reside in the CLASSPATH and coexist with the Thin Client for JAX-WS.

Before you set up a JAX-WS unmanaged client execution environment, obtain the Thin Client for JAX-WS Java archive (JAR) file. To obtain the Thin Client for JAX-WS, install WebSphere Application Server Version 8.5 or the Application Client for WebSphere Application Server Version 8.5. The Thin Client for JAX-WS JAR file, `com.ibm.jaxws.thinclient_8.5.0.jar`, is located in the *app_server_root*`\runtimes` directory.

Copy the Thin Client for JAX-WS, `com.ibm.jaxws.thinclient_8.5.0.jar` file and the `endorsed_apis_8.5.0.jar` files, to other machines to create a lightweight client environment that enables communications with the product. Copies of the Thin Client for JAX-WS are subject to the same terms and conditions of the license agreement for the WebSphere product where you obtained the Thin Client for JAX-WS. Refer to the license agreements for correct usage and other limitations.

The Thin Client for JAX-WS is supported in the following environments:

- IBM Software Development Kits (SDKs) Version 6.0
- non-IBM SDKs V6.0 with the following limitation:
  - Xerces limitation on non-IBM SDKs

    You must download Xerces-J Version 2.6.2, and add the file to the classpath when setting up the Thin Client for JAX-WS environment.
  - WS-SecurityKerberos on non-IBM SDKs

    WS-SecurityKerberos is not supported with the Sun JDK or other non-IBM SDKs. Applications running in a Thin Client for JAX-WS environment that make use of WS-Security message level protection and use Kerberos security tokens as described in the Web Services Security Kerberos Token Profile 1.1 specification, do not correctly work on non-IBM JDKs. This limitation exists because of a dependancy on the IBM JGSS provider that is only available within IBM SDKs.
- Equinox 3.6 OSGi runtime environments

About this task

Set up a Thin Client for JAX-WS environment by completing the following steps.

Procedure

1. Copy the Thin Client for JAX-WS JAR file, `com.ibm.jaxws.thinclient_8.5.0.jar`, to other machines to create a lightweight client environment.
2. Use the Java Endorsed Standards Override Mechanism to override APIs that are available in the JDK on your system.

   Because the Thin Client for JAX-WS with WebSphere Application ServerVersion 8.5 requires APIs that are more current than what is available in JDKs to support JAX-WS 2.2 and JAXB 2.2 implementations, you must override the default JDK APIs in use by your system by using the Java Endorsed Standards Override Mechanism.

   Copy the *app_server_root*`\runtimes\endorsed\endorsed_apis_8.5.0.jar` file into the default directory, `JAVA_JRE\lib\endorsed`. Alternatively, you can use the `java.endorsed.dirs` property to specify a directory of your choice. If you choose to use an alternative directory, it is a best practice to only include the `endorsed_apis` JAR file.
3. Configure the path. Enter the following command to add the Java `bin` directories to your path:

```
set PATH=<your_JDK_bin_directory>;%PATH%
```

4. Configure the classpath.
   - Add the Thin Client for JAX-WS JAR file to the classpath definition.

     Important: If the Thin Client is to use the Java Message Service (JMS), then **all** the JAR files that are required must be in the classpath for JMS and for the client so that entries exist for all the required files. Otherwise, required files will not be identified as installed and ready for use.

```
set CLASSPATH=.;<your_jax-ws_thin_client_install_directory>\com.ibm.jaxws.thinclient_8.5.0.jar;
<your_application_jars>;%CLASSPATH%
```

   - If you are using a non-IBM SDK, obtain a Xerces `xml-apis.jar` file and `xercesImpl.jar` file from the Xerces website, and configure the classpath definition.

```
set CLASSPATH=.;<your_Xerces_install_directory>\xml-apis.jar;<your_Xerces_install_directory>
\xercesImpl.jar;%CLASSPATH%
```

5. Optional: Implement policy sets for your client.
6. Configure SSL for the client.
   a. Add the following system properties to the Java command:

```
-Dcom.ibm.SSL.ConfigURL=file:///home/sample/ssl.client.props
```

   You can obtain the `ssl.client.props` file from the WebSphere Application Server installation and modify the file to suit your environment. You must, at a minimum, update the location of the `com.ibm.ssl.keyStore` and `com.ibm.ssl.trustStore` key files in the `ssl.client.props` file to the match location of your target environment. For example, use these SSL configuration settings when running the application with a Sun JRE:

```
com.ibm.ssl.protocol=SSL
com.ibm.ssl.trustManager=SunX509
com.ibm.ssl.keyManager=SunX509
com.ibm.ssl.contextProvider=SunJSSE

com.ibm.ssl.keyStoreType=JKS
com.ibm.ssl.keyStoreProvider=SUN
com.ibm.ssl.keyStore=/home/user1/etc/key.jks

com.ibm.ssl.trustStoreType=JKS
com.ibm.ssl.trustStoreProvider=SUN
com.ibm.ssl.trustStore=/home/user1/etc/trust.jks
```

   The key store file and trust store file must be created using the Java keytool utility before the application runs. The automatic key file generation is not supported with a non-IBM product JRE.

7. Run your client application:
   - Enter the following command if you have copied the `endorsed_apis_8.5.0.jar` file into the `JAVA_JRE\lib\endorsed` default directory; for example:

```
%JAVA_HOME%\bin\java -Dcom.ibm.SSL.ConfigURL=file:\\\home\sample\ssl.client.props <your_client_application>
```

   - Enter the following command if you have copied the `endorsed_apis_8.5.0.jar` file into a directory other than the default `JAVA_JRE\lib\endorsed` directory; for example:

```
%JAVA_HOME%\bin\java
-Djava.endorsed.dirs=<directory_that_includes_endorsed_apis_8.5.0.jar>
-Dcom.ibm.SSL.ConfigURL=file:\\\home\sample\ssl.client.props <your_client_application>
```

Results

You have set up an unmanaged JAX-WS client runtime environment to invoke web services hosted on a WebSphere Application Server.

Testing web services-enabled clients

Once you have developed, assembled, deployed and configured your web service, you can test to confirm your web service runs in the application server environment.

Before you begin

Before testing your web services Java client to confirm your web service runs in the WebSphere Application Server environment, verify that the server endpoint specified in the client Web Services Description Language (WSDL) file is running and available.

About this task

Tests are run differently depending on whether the client module is in a Java EE container or if the client is running in the Thin Client for Java API for XML-based RPC (JAX-RPC) with WebSphere Application Server application environment or the Thin Client for Java API for XML-Based Web Services (JAX-WS) with WebSphere Application Server application environment.

Procedure

1. Test an unmanaged client JAR file by running your application with the **java** command.

   For JAX-WS applications:

```
"$JAVA_HOME/bin/java"
-Djava.endorsed.dirs=<your_jax-ws_thin_client_install_directory>/endorsed_apis_8.5.0.jar
-classpath
"<your_JAX-WS_thin_client_install_directory>/runtimes/com.ibm.jaxws.thinclient_8.5.0.jar:
<list_of_your_application_jars_and_classes>"
<fully_qualified_class_name_to_run> <your_application_parameters>
```

   For JAX-RPC applications:

```
"$JAVA_HOME/bin/java"
-classpath
"<your_JAX-RPC_thin_client_install_directory>/runtimes/com.ibm.ws.webservices.thinclient_8.5.0.jar:
<list_of_your_application_jars_and_classes>"
<fully_qualified_class_name_to_run> <your_application_parameters>
```

   The unmanaged client application runs.

2. Test a managed JAX-RPC client EAR file or an unmanaged JAX-WS client EAR file.

   a. Run your client application with the **launchClient** command. The following example illustrates the use of this command:

```
launchClient clientEar
```

Results

You have a web services-enabled client that is tested. Now you can add security measures to the web service. Security measures are optional.

Chapter 49. Deploying web services - RESTful services

You can use Java API for RESTful Web Services (JAX-RS) to develop services that follow Representational State Transfer (REST) principles. RESTful services are based on manipulating resources. Resources can contain static or dynamically updated data. By identifying the resources in your application, you can make the service more useful and easier to develop.

Deploying JAX-RS web applications

After you assemble your Java API for RESTful Web Services (JAX-RS) web application, you can deploy the web application archive (WAR) package or the enterprise archive (EAR) package onto the application server.

Before you begin

To deploy a JAX-RS web application, you need a WAR package or EAR package that is configured and enabled for RESTful services.

About this task

Every web application must have a context root for the web application to deploy successfully. A context root for each Web module is defined in the application deployment descriptor during application assembly or during application deployment. The context root is combined with the defined servlet mapping from the WAR file to compose the full URL that users type to access the servlet. The context root for each deployed web application must be unique on the server. The context root can also be empty. For instance, if a Web application used a context root of `sample/application/`, the web application request URL begins with `http://<hostname>:<port>/sample/application/`.

The URL pattern of a servlet is appended to the context root of the Web application. For example, if the context root is `sample/application/` and the servlet URL mapping is `rest/api/*`, the base URI for the JAX-RS web application is `http://<hostname>:<port>/sample/application/rest/api`.

Procedure

Deploy the JAX-RS web application WAR package or EAR package onto the application server. Read about installing enterprise application files to learn more about deploying web applications.

Results

The JAX-RS web application is deployed and ready for your business use.

Deployment of a Java API for RESTful Web Services (JAX-RS) web application is successful if you can access the application by typing a Uniform Resource Locator (URL) in a browser or if you can access the application by following a link. If you cannot access your application, follow these steps to eliminate some common errors that can occur during deployment.

Note:

Use the following tips to resolve common errors during deployment of JAX-RS web applications.

An HTTP 404 Not Found error message is sent back to the client in the server response.

To resolve this problem, take the following actions:

- Verify that the root resource classes are annotated with a @javax.ws.rs.Path annotation and that value in the annotation is correct. Root resource classes without a @Path

annotation are not registered with the JAX-RS runtime. To learn more, see the defining the URI patterns for resources in RESTful applications information.

- Verify that the root resource class is added to the set of classes returned from the getClasses() method for the subclasses of the javax.ws.rs.core.Application class. Classes not registered in the subclasses of the javax.ws.rs.core.Application class are not recognized by the JAX-RS runtime environment. To learn more, see the defining the URI patterns for resources in RESTful applications information.
- Verify that the web.xml configuration is correct with the expected URL patterns. For additional details, see the configuring the web.xml file for JAX-RS servlets and filters information.
- Verify that the URL that is being used is correct and includes the context root. If you are using a servlet, the servlet URL pattern is a part of the final URL. Using a filter might be more suitable in your web application. For additional details, see the configuring the web.xml file for JAX-RS servlets and filters information.

An HTTP 406 Not Acceptable error message is automatically being sent back to the client in the server response.

To resolve this problem, take the following actions:

- Verify that the Accept HTTP request header in the incoming request is correct. To learn more, see the Implementing content negotiation based on HTTP headers information.
- Verify that the @javax.ws.rs.Produces value on the resource method or resource class is compatible with the incoming Accept HTTP request header. To learn more, see the defining media types for resources in RESTful applications.

An HTTP 415 Unsupported Media Type error message is automatically being sent back to the client in the server response.

To resolve this problem, take the following actions:

- Verify that that the Content-Type HTTP request header in the incoming request is correct and is being sent. To learn more, see the defining media types for resources in RESTful applications.
- Verify that the @javax.ws.rs.Consumes value on the resource method or resource class is compatible with the incoming Content-Type HTTP request header.

An HTTP 204 No Content response status is automatically being sent back to the client in the server response.

To resolve this problem, take the following action:

- If the object returned from a resource method is a null value, then a 204 No Content status response code is sent back from the server runtime automatically.

For information about known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem.

Chapter 50. Deploying web services - Security (WS-Security)

The Web Services Security specification defines core facilities for protecting the integrity and confidentiality of a message, and provides mechanisms for associating security-related claims with a message.

Deploying applications that use SAML

After SAML policy sets and bindings have been configured, and SAML tokens created, the SAML token information can be sent from the original login server to other servers using the SAML propagation feature. You can also extract SAML attributes from an existing SAML token and then create additional tokens using the extracted attributes.

About this task

Use the SAML propagation feature of WebSphere Application Server to send SAML token information based on the original login to other servers using a SAML token. Four propagation methods are provided. You can propagate the original SAML token, the SAML token identity and attributes, the WSCredential and WSPrincipal information, or a pre-existing SAML token inserted in the RequestContext.

When SAML is installed on a WebSphere server, you can create SAML attributes using the SAML runtime API. The SAML attributes are added to a CredentialConfig object, which is used to generate a SAML token. The API also provides a function that extracts SAML attributes from an existing SAML token and processes the attributes.

The following topics provide more information about deploying SAML applications.

Propagating SAML tokens

You can use various SAML token propagation methods to include SAML tokens in outbound web services messages.

About this task

A web services client can include two types of tokens in outbound web services messages:

- Original SAML tokens the client received from inbound web services messages.
- New self-issued SAML tokens.

New SAML tokens can be generated using attributes from the original SAML tokens, or using attributes from the WSPrincipal user name in the RunAs Subject. The web services policy configuration determines which SAML tokens will be propagated. You can override the policy configuration by programmatically inserting SAML tokens that you want to propagate into the Axis2 RequestContext object.

Four propagation methods are enabled. This table summarizes the propagation methods and the associated binding options:

Table 289. Propagation methods and associated binding options. Use propagation to include SAML tokens in web services messages.

SAML token propagation method	Binding option	Implementation details
Propagate the original SAML token.	The tokenRequest binding option is set to the value, `propagation`.	Sends the original SAML token from the server where the token was received to other servers using WS-Security.
Propagate the user security name, unique security name, group IDs, and security realm name.	The tokenRequest binding option is set to the value, `issueByWSCredential`.	Overrides the default system implementation. The self-issued SAML token contains user security name, user unique security name, group IDs, and security realm name that are specified by the WSCredential object in user security context.

Table 289. Propagation methods and associated binding options (continued). Use propagation to include SAML tokens in web services messages.

SAML token propagation method	Binding option	Implementation details
Propagate the SAML token identity and attributes.	No binding option is set.	Default system implementation. The server self-generates a new SAML token containing the original SAML attributes, Authentication method, and NameIdentifier or SAML NameID, and sends the new self-generated SAML token to downstream servers using WS-Security. The new SAML token issuer name, issuer signing certificate, and lifetime are determined by the SAML provider configuration properties.
Propagate the WSPrincipal.	The tokenRequest binding option is set to the value, `issueByWSPrincipal`.	Overrides the default system implementation. The self-issued SAML token contains WSPrincipal information in the RunAs subject. The information is stored as NameIdentity or NameID without copying anything from the original SAML token, even if the token exists in the subject.
Programmatically propagate a pre-existing SAML token.	Insert the SAML token that you want to propagate into the RequestContext using the property, `com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.SAMLTOKEN_IN_MESSAGECONTEXT`.	Overrides all other existing binding options.

Procedure

1. Propagate the original SAML token by setting the tokenRequest binding option to the value, `propagation`, in the bindings.xml file, as shown in the steps. This method sends the original SAML token to other servers using WS-Security. In order for the propagation to succeed, there must be a valid SAML token in the RunAs subject. The server extracts the SAML token from the RunAs subject in the current security context and validates the following conditions. If any of these conditions are invalid, the WS-Security runtime environment does not propagate the SAML token, and the propagation request fails.
   - The SAML token has not expired, and the expiration time is within the time window of the notOnOrAfter value.
   - The ConfirmationMethod setting in the SAML token is the same as the confirmationMethod binding option defined in the token generator configuration.
   - The token ValueType in the SAML token matches the ValueType in the token generator configuration.

   Perform these steps to set the correct value for the tokenRequest binding option. This procedure assumes that a web services client application named JaxWSServicesSamples is deployed, and that the Saml Bearer Client sample binding is attached.
   a. Click **Applications > Application types > WebSphere enterprise Applications > *JaxWSServicesSamples* > Service client policy sets and bindings > *Saml Bearer Client sample* > WS-Security > Authentication and protection**.
   b. Click **gen_saml11token** in the Authentication tokens table.
   c. Click **Callback handler**.
   d. Add the custom property `tokenRequest` and set the property value to `propagation`.
2. To propagate the SAML token identity and attributes using a self-issuing SAML token, modify the outbound tokenGenerator in the bindings.xml file. This method sends the original SAML attributes, NameIdentifier or NameID, and authentication method from the original SAML token to other servers using WS-Security. If there is no SAML token in the subject, the server uses the WSPrincipal, stored as NameIdentifier or NameID, to create a self-issued SAML token. This propagation method is the default system implementation. In this method, the binding option is not set.

   The following limitations apply to the bindings.xml file when you are using this propagation method:
   - Do not set the tokenRequest binding option in the bindings.xml file.
   - Do not set the stsURI binding option in the bindings.xml file, or set the option to this value: `www.websphere.ibm.com/SAML/Issuer/Self`.

3. To propagate the WSPrincipal, modify the bindings.xml file as shown in the steps. Set the tokenRequest binding option to the value, `issueByPrincipal`, in the bindings.xml file. Using this method, the self-issued SAML token is always based on the WSPrincipal even if there is a SAML token in the subject. The new SAML token contains the WSPrincipal user name as the NameId or NameIdentifier. The token does not contain any other attributes in the WSPrincipal or WSCredential objects.

   The following limitation applies to the bindings.xml file when you are using this propagation method:

   - Do not set the stsURI binding option in the bindings.xml file, or set the option to the value, `www.websphere.ibm.com/SAML/Issuer/Self`.

   Perform these steps to set the correct value for the tokenRequest binding option. This procedure assumes that a web services client application named JaxWSServicesSamples is deployed, and that the Saml Bearer Client sample binding is attached.

   a. Click **Applications > Application types > WebSphere enterprise Applications > *JaxWSServicesSamples* > Service client policy sets and bindings > *Saml Bearer Client sample* > WS-Security > Authentication and protection**.

   b. Click **gen_saml11token** in the Authentication tokens table.

   c. Click **Callback handler**.

   d. Add the custom property `tokenRequest` and set the property value to `issueByPrincipal`.

4. To propagate a pre-existing SAML token by inserting SAMLToken in the RequestContext, follow these steps. Use this method to send a SAML token that you created to downstream servers using WS-Security. The propagation is automatically triggered if the WS-Security runtime detects a SAML token in the RequestContext. The pre-existing token overrides any other existing binding options. To use this propagation method, save the existing SAML token in the RequestContext by specifying `com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.`*`SAMLTOKEN_IN_MESSAGECONTEXT`* as the key, as shown in the steps.

   a. Generate a SAML token using the method `SAMLToken samlToken = <token type>`, for example:

```
SAMLToken samlToken = samlFactory.newSAMLToken(cred, reqData, samlIssuerCfg);
```

   b. Save the SAMLToken to the RequestContext, for example:

```
Map requestContext = ((BindingProvider)port).getRequestContext();
   requestContext.put("com.ibm.wsspi.wssecurity.core.token.config.WSSConstants.SAMLTOKEN_IN_MESSAGECONTEXT", samlToken );
```

   This propagation option can co-exist with the other propagation methods, and overrides the other methods. If the SAML token in the RequestContext is expired, or the token expiration time is less than current time plus the cache cushion, the WS-Security runtime environment ignores the SAML token, and uses one of the other three propagation methods that is configured in the bindings.xml file. To avoid using the other three propagation methods, add the following binding option to the custom properties under callback handler in the TokenGenerator configuration: `failOverToTokenRequest = false`.

5. To propagate a user's group memberships, unique security name, and realm name contained in the `com.ibm.websphere.security.cred.WSCredential` object, modify the `bindings.xml` file as shown in the steps. Set the `tokenRequest` binding option to the value, `issueByWSCredential`, in the `bindings.xml` file. Using this method, the self-issued SAML token is always based on the `WSCredential` even if there is a SAML token in the subject.

   The new SAML 1.1 token contains the following assertions:

   - The `NameIdentifier` element contains the `SecurityName` value from `WSCredential` with the `NameQualifier` element set to the realm name from `WSCredential`. The `SecurityName` is obtained by calling the `WSCredential.getSecurityName()` method. The realm name is obtained by calling the `WSCredential.getRealmName()` method.
   - All attributes have an `AttributeNamespace` set to `com.ibm.websphere.security.cred.WSCredential` as the value.
   - The `GroupIds` attribute contains all group names that a user belongs to. The group names are obtained by calling the `WSCredential.getGroupIds()` method.

- The UniqueSecurityName attribute contains the unique security name, which is obtained by calling the WSCredential.getUniqueSecurityName() method.
- Optionally, you can assert the realm name from WSCredential by adding the includeRealmName=true custom property in the callback handler.

The new SAML 2.0 token contains the following assertions:

- The NameID element contains the SecurityName value from WSCredential with the NameQualifier element set to the realm name from WSCredential. The SecurityName is obtained by calling the WSCredential.getSecurityName() method. The realm name is obtained by calling the WSCredential.getRealmName() method.
- All attributes have a NameFormat set to com.ibm.websphere.security.cred.WSCredential as the value.
- The GroupIds attribute contains all group names that a user belongs to. The group names are obtained by calling the WSCredential.getGroupIds() method.
- The UniqueSecurityName attribute contains the unique security name, which is obtained by calling the WSCredential.getUniqueSecurityName() method.
- Optionally, you can assert the realm name from WSCredential by adding the includeRealmName=true custom property in the callback handler.

The following limitation applies to the bindings.xml file when you use the propagation method:

- Do not set the stsURI binding option in the bindings.xml file.

Perform these steps to set the correct value for the tokenRequest binding option. This procedure assumes that a Web services client application named JaxWSServicesSamples is deployed, and that the Saml Bearer Client sample binding is attached.

a. Click **Applications > Application types > WebSphere enterprise Applications > *JaxWSServicesSamples* > Service client policy sets and bindings > *Saml Bearer Client sample* > WS-Security > Authentication and protection.**
b. Click **gen_saml11token** in the Authentication tokens table.
c. Click **Callback handler**.
d. Add the tokenRequest custom property and set the property value to issueByWSCredential.

The following example illustrates the NameIdentifier and Attribute statement from a self-issued SAML 1.1 assertion based on WSCredential.

```
<saml:AttributeStatement>
    <saml:Subject>
        <saml:NameIdentifier NameQualifier="ldap.acme.com:9080">uid=alice,dc=acme,dc=com</saml:NameIdentifier>
        <saml:SubjectConfirmation>
            <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
        </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Attribute AttributeName="UniqueSecurityName" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>uid=alice,dc=acme,dc=com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute AttributeName="GroupIds" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>cn=development,dc=acme,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=deployment,dc=acme,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=test,dc=acme,dc=com</saml:AttributeValue>
    </saml:Attribute>
</saml:AttributeStatement>
```

The following example illustrates the NameID and Attribute statement from a self-issued SAML 2.0 assertion based on WSCredential.

```
<saml2:AttributeStatement>
    <saml2:Attribute Name="UniqueSecurityName"
           NameFormat="com.ibm.websphere.security.cred.WSCredential">
        <saml2:AttributeValue>uid=alice,dc=acme,dc=com</saml2:AttributeValue>
    <saml2:Attribute>
    <saml2:Attribute AttributeName="GroupIds"
           NameFormat="com.ibm.websphere.security.cred.WSCredential">
        <saml2:AttributeValue>cn=development,dc=acme,dc=com</saml2:AttributeValue>
```

```
        <saml2:AttributeValue>cn=deployment,dc=acme,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=test,dc=acme,dc=com</saml2:AttributeValue>
    </saml2:Attribute>
</saml2:AttributeStatement>

<saml2:NameID NameQualifier="ldap.acme.com:9060">alice</saml2:NameID>
```

Creating SAML attributes in SAML tokens

Using the SAML runtime API, you can create SAML tokens containing SAML attributes. You can also extract the SAML attributes from an existing SAML token.

About this task

Using WebSphere Application Server, you can create SAML attributes using the SAML token library APIs. The SAML attributes are added to a CredentialConfig object, which is used to generate a SAML token. The API also provides a function that extracts SAML attributes from an existing SAML token and processes the attributes.

To create a SAML token containing SAML attributes, perform the following steps:

Procedure

1. Initialize a com.ibm.wsspi.wssecurity.saml.data.SAMLAttribute object. This creates a SAML attribute based on an address, for example:

```
SAMLAttribute sattribute =
         new SAMLAttribute("urn:oid:2.5.4.20", //Name
       new String[] {" any address"}, //Attribute Values
       null,  /*XML Attributes empty on this example*/
       "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500", //NameSpace
       "urn:oasis:names:tc:SAML:2.0:attrname-format:uri",  //format
       "Address");
```

2. Use the SAMLTokenFactory to create a CredentialConfig object containing a SAML attribute. This method requires the Java security permisson wssapi.SAMLTokenFactory.newCredentialConfig.
   a. Create a com.ibm.wsspi.wssecurity.saml.config.CredentialConfig object and set a valid principal name.
   b. Create a SAML attribute.
   c. Create a list of SAML attributes and add the SAML attribute to the list.
   d. Add the SAML attribute list to the CredentialConfig object.

   See the following example:

```
SAMLTokenFactory samlFactory =
  SAMLTokenFactory.getInstance("http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0");//samlTokenType

CredentialConfig credentialConfig = samlFactory.newCredentialConfig();
credentialConfig.setRequesterNameID("any name");

SAMLAttribute sattribute =
         new SAMLAttribute("urn:oid:2.5.4.20", //Name
       new String[] {" any address"}, //Attribute Values
       null,  /*XML Attributes empty on this example*/
       "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500", //NameSpace
       "urn:oasis:names:tc:SAML:2.0:attrname-format:uri",  //format
       "Address");

ArrayList<SAMLAttribute> al = new ArrayList<SAMLAttribute>();
al.add(sattribute);
credentialConfig.setSAMLAttributes(al);
```

3. Specifying the CredentialConfig as a parameter, use the com.ibm.websphere.wssecurity.wssapi.token.SAMLTokenFactory newSAMLToken method to create a SAML token containing the attributes. This step assumes that a RequesterConfig `reqData` object and a ProviderConfig `samlIssuerCfg` object have already been created. For more information on these objects, read about RequesterConfig and ProviderConfig.
   a. Obtain an instance of the SAMLTokenFactory.
   b. Create a SAML token using the newSAMLToken method from the SAMLTokenFactory, for example:

```
SAMLTokenFactory samlFactory =
  SAMLTokenFactory.getInstance("http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1");

SAMLToken aSamlToken = samlFactory.newSAMLToken(credentialConfig, reqData, samlIssuerCfg);
```

4. Optional: Extract SAML attributes from an existing SAML token. This step is useful to extract the SAML attributes from a received SAML token. You can use this step when a SAML assertion is received and the attributes contained in the assertion need to be processed.
   a. Invoke the getSAMLAttributes() method with the token as a parameter to obtain a list of the SAML attributes in the token. This method requires the Java security permission wssapi.SAMLToken.getSAMLAttributes.
   b. Apply an iterator to the list.
   c. Iterate through the list and perform any additional processing required for your application.

   See the following example:

```
List<SAMLAttribute> aList = aSAMLToken.getSAMLAttributes();
java.util.Iterator<SAMLAttribute> i = aList.iterator();

while(i.hasNext()){

   SAMLAttribute anAttribute = i.next();

   //do something with namespace
   String namespace = anAttribute.getAttributeNamespace();

   //do something with name
   String name = anAttribute.getName();

   //do something with friendly name
   String friendlyName = anAttribute.getFriendlyName();

   //process sring attribute values
   String[] stringAttributeValues = anAttribute.getStringAttributeValue();

   //process XML attribute values
   XMLStructure[] xmlAttributeValues = (XMLStructure[]) anAttribute.getXMLAttributeValue();

   }
```

SAML user attributes

A SAML assertion can contain user attributes relating to the principal of the SAML token. A SAML assertion can contain multiple user attributes.

You can include user attributes in the token to communicate the address of the person who is the SAML assertion principal. This example shows a SAML assertion containing a user attribute:

```
<saml:AttributeStatement>
  <saml:Attribute  xmlns:x500=
      "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
    NameFormat=
      "urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
    Name="urn:oid:2.5.4.20"
    FriendlyName="Address">
          <saml:AttributeValue xsi:type="xs:string">
                      11111 Parker Lane, Austin, Texas, 78758
          </saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

This table describes the parameters used in the assertion:

Parameter	Description
NameFormat	Specifies how the attribute is interpreted.
Name	Indicates the formal name of the attribute.
FriendlyName	Provides a user-friendly name for an attribute when the Name parameter is cryptic.

Parameter	Description
AttributeValue	The value of the user attribute. The value can be a string, or a complex XML type.

Establishing security context for web services clients using SAML security tokens

WebSphere Application Server supports two policy set caller binding configuration options to establish client security context using SAML security tokens in web services SOAP request messages. The two configuration options are mapping SAML tokens to a user entry in a local user repository and, asserting SAML tokens based on a trust relationship.

Before you begin

This task assumes that you are familiar with WebSphere Application Server SAML technology.

About this task

This task describes setting the WebSphere Application Server policy set caller binding configuration option to establish client security context using SAML security tokens in web services SOAP request messages. You can either map SAML tokens to a user entry in a local user repository or assert SAML tokens based on a trust relationship. The second configuration option does not require accessing the local user repository. Instead, the WS-Security runtime environment populates the client security context entirely using the contents of SAML security tokens. This process is based on a trust relationship to the SAML token issuer. If a SAML tokens specifies the sender-vouches subject confirmation method. the process is based on a trust relationship to the message sender.

Procedure

1. Configure a policy set caller binding and select the SAML token type to represent a web services client request.
   a. Click **WebSphere enterprise applications** > ***application_name*** > **Service provider policy sets and bindings** > ***binding_name*** > **WS-Security** > **Callers**.
   b. Click **New** to create the caller configuration.
   c. Specify a **Name**, such as `caller`.
   d. Enter a value for the **Caller identity local part**. For example, `http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0`, which must match the local part of the `CustomToken` element in the attached WS-Security policy.
   e. Click **Apply** and **Save**.
2. Optional: Map SAML security tokens to a user entry in a local user repository. Mapping to a user entry is the default behavior when you configure a caller binding without specifying a configuration option. Alternatively and optionally, you can select this configuration option explicitly using the following steps:
   a. On the caller binding configuration page, add a Callback handler: `com.ibm.websphere.wssecurity.callbackhandler.SAMLIdAssertionCallbackHandler`.
   b. Add a Callback handler custom property, `crossDomainIdAssertion`, and set its value to `false`.
3. Optional: Assert SAML security tokens based on trust relationship.
   a. On the caller binding configuration page, add a Callback handler: `com.ibm.websphere.wssecurity.callbackhandler.SAMLIdAssertionCallbackHandler`.
   b. Add a Callback handler custom property, `crossDomainIdAssertion`, and set its value to `true`.

   In WebSphere Application Server Version 7.0 Fix Pack 7 and later releases, the WS-Security runtime environment takes a SAML token Issuer name to represent the foreign security realm name. WS-Security takes the `NameID` element in the case of SAML 2.0 security tokens or the `NameIdentifier`

element in the case of SAML 1.1 security tokens to represent user security name. Alternatively, you can explicitly specify which SAML token attribute to use to represent user security name. Moreover, you can also specify which SAML token attribute to use to represent user group membership. Read about SAML assertions across WebSphere Application Server security domains for a detailed discussion of the SAML token assertion trust model and binding configuration.

Version 8.x supports propagating select security context data in SAML tokens. You must set a `tokenRequest` custom property with an `issueByWSCredential` property value in the WS-Security binding configuration of the web services client. Read about propagating SAML tokens for a detailed description of this binding option. When the `crossDomainIdAssertion` property is set to `true` in Version 8.x, WS-Security checks whether a SAML token contains a SAML Attribute `UniqueSecurityName` with a `NameFormat` element with a value of `com.ibm.websphere.security.cred.WSCredential`. If found, WS-Security uses the `NameQualifier` attribute value of the `NameID` element or `NameIdentifier` element to represent the user security realm name. WS-Security also uses the `UniqueSecurityName` attribute value and the `GroupIds` attribute value to represent a unique user name and group membership. This default behavior is different between Version 7 and Version 8.x of the product. You can add a `CallbackHandler` property, `IssuerNameForRealm`, and set its value to `true` to configure Version 8.x to preserve the Version 7 behavior. Alternatively, you can add a `CallbackHandler` property, `NameQualifierForRealm`, and set its value to `true` to configure Version 8.x to always use the `NameQualifier` attribute to represent the user security realm name.

Results

You have configured a web service to establish a client security context using the SAML security token in the web services SOAP request messages.

Example

The following example illustrates the `NameIdentifier` and `Attribute` elements from a self-issued SAML 1.1 assertion based on `WSCredential`:

```
<saml:AttributeStatement>
    <saml:Subject>
        <saml:NameIdentifier NameQualifier="ldap.example.com:9080">uid=alice,dc=example,dc=com</saml:NameIdentifier>
        <saml:SubjectConfirmation>
            <saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:bearer</saml:ConfirmationMethod>
        </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Attribute AttributeName="UniqueSecurityName" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>uid=alice,dc=example,dc=com</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute AttributeName="GroupIds" AttributeNamespace="com.ibm.websphere.security.cred.WSCredential">
        <saml:AttributeValue>cn=development,dc=example,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=deployment,dc=example,dc=com</saml:AttributeValue>
        <saml:AttributeValue>cn=test,dc=example,dc=com</saml:AttributeValue>
    </saml:Attribute>
</saml:AttributeStatement>
```

The following example illustrates the `NameID` and `Attribute` elements from a self-issued SAML 2.0 assertion based on `WSCredential`:

```
<saml2:AttributeStatement>
    <saml2:Attribute Name="UniqueSecurityName" NameFormat="com.ibm.websphere.security.cred.WSCredential" />
    <saml2:AttributeValue>uid=alice,dc=example,dc=com</saml2:AttributeValue>
    <saml2:Attribute>
        <saml2:Attribute AttributeName="GroupIds" NameFormat="com.ibm.websphere.security.cred.WSCredential" />
        <saml2:AttributeValue>cn=development,dc=example,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=deployment,dc=example,dc=com</saml2:AttributeValue>
        <saml2:AttributeValue>cn=test,dc=example,dc=com</saml2:AttributeValue>
    </saml2:Attribute>
<saml2:AttributeStatement>

<saml2:NameID NameQualifier="ldap.example.com:9060">alice</saml2:NameID>
```

Chapter 51. Deploying web services - Transports

Transport chains represent a network protocol stack that is used for I/O operations within an application server environment. Transport chains are part of the channel framework function that provides a common networking service for all components.

Invoking JAX-WS web services asynchronously using the HTTP transport

Using the JAX-WS asynchronous response servlet

Java API for XML-Based Web Services (JAX-WS) includes an asynchronous response servlet, which is used within the application server environment to receive responses for JAX-WS requests that are invoked asynchronously.

Before you begin

JAX-WS provides support for invoking web services using an asynchronous client invocation by using either a callback or polling model. Both the callback model and the polling model are available on the Dispatch client and the dynamic proxy client. When a JAX-WS client that is running within the application server environment uses an asynchronous client invocation, the responses are received by the asynchronous response servlet. To learn how to use the asynchronous client invocation model, read about invoking JAX-WS web services asynchronously.

About this task

The asynchronous response servlet is used within an application server to handle incoming asynchronous responses. The servlet uses the same secure and unsecure HTTP ports assigned to the application server. The servlet starts automatically when the application server starts. Because the asynchronous response servlet does not perform role-based authorization checks, only user authentication checks are performed.

The asynchronous response servlet supports both the HTTP and HTTPS protocols. Since the servlet inherits the SSL configuration of the application server, configuring the application server also configures the servlet. The asynchronous response servlet is not affected by the custom HTTP and SSL port properties used by the asynchronous response listener and only runs on the application ports for the application server.

Procedure

1. Determine if you want the JAX-WS client to use the HTTP or HTTPS transport mechanism.
2. Configure the web container transport chains to modify the SSL configuration of the application server. The servlet inherits these settings. Read about configuring transport chains to learn how to configure the web container transport chains.

Results

The asynchronous response servlet is configured to enable your JAX-WS clients to receive asynchronous responses on the HTTP or HTTPS transport protocol.

Note: When you add a new application server to your environment, the asynchronous response servlet is automatically restarted so the deployment.xml file can be updated for the new application server. If your application receives an incoming response when the asynchronous response servlet is restarting, the incoming response might fail with an HTTP 404 error.

Note: JAX-WS services do not successfully return asynchronous responses to clients that are installed in application security-enabled WebSphere Application Servers. Because the Asynchronous Response Servlet for WebSphere Application Server, which handles asynchronous web services responses, is protected when application security is enabled, you must supply a credential together with the JAX-WS service incoming response. Attach the HTTPTransport policy set binding to the JAX-WS service in the service attachment. Additionally, enter a valid basic authentication user ID and password, which are defined in the user registry of the client, into the Basic authentication for outbound asynchronous service responses field.

Using the JAX-WS asynchronous response listener

Java API for XML-Based Web Services (JAX-WS) includes an asynchronous response listener, which is used within the Thin Client for JAX-WS and application client environments to receive responses for requests that are invoked asynchronously.

Before you begin

JAX-WS provides support for invoking web services using an asynchronous client invocation by using either a callback or polling model. Both the callback model and the polling model are available on the Dispatch client and the dynamic proxy client. When the JAX-WS client uses an asynchronous client invocation, the responses are received by the asynchronous response listener. To learn how to use the asynchronous client invocation model, read about invoking JAX-WS web services asynchronously.

About this task

The asynchronous response listener is used within a Web services client to handle incoming asynchronous responses. You can use the listener in Thin Client for JAX-WS environments and application client environments. By default, the listener opens a random port to listen for asynchronous responses or you can optionally configure a specific port for the listener to use. The listener starts automatically in the JAX-WS run time when the JAX-WS client is configured to expect an asynchronous response.

There are two versions of the asynchronous response listener. The unsecure version of the asynchronous response listener supports the HTTP protocol, and the secure version of the asynchronous response listener supports the HTTPS protocol. The correct asynchronous response listener is automatically started based on the particular transport used by the JAX-WS client. To ensure that the correct Secure Sockets Layer (SSL) handshaking occurs between the asynchronous response listener and the application server, configure the SSL properties using the SSL transport policy or the Java system properties.

For web services clients running in the application server environment, use the asynchronous response servlet for receiving asynchronous responses.

Procedure

1. Determine if you want the JAX-WS client to use the HTTP or HTTPS transport mechanism.
2. Configure the asynchronous response listener for unsecure communication using HTTP.

   You can configure the HTTP port for the asynchronous response listener as a Java system property or as a custom property within the transport policy. Properties that are defined in the policy set binding files override any Java system property that might have been defined.

   a. Define the `com.ibm.websphere.webservices.http.listenerPort` property as a Java system property. If this property is set as a Java system property, then all asynchronous response listeners within that Java Virtual Machine (JVM) are affected.

   b. Define the `com.ibm.websphere.webservices.http.listenerPort` property within the `HTTPTransport` transport policy set bindings files. If this property is set as a custom property within a transport policy set binding, then only the services for which the policy set has been configured are affected.
3. Configure the asynchronous response listener for secure communication using HTTPS.

You can configure the HTTPS port for the asynchronous response listener as a Java system property or as a custom property within the transport policy.

a. Define the `com.ibm.websphere.webservices.https.listenerPort` property as a Java system property. If this property is set as a Java system property, thenall asynchronous response listeners within that JVM are affected.
b. Define the `com.ibm.websphere.webservices.https.listenerPort` property within the `SSLTransport` transport policy set bindings files. If this property is set as a custom property within a transport policy set binding, then only the services for which the policy set has been configured are affected.

Results

Your JAX-WS web services client is configured to use the asynchronous response listener to receive incoming asynchronous responses.

Example

The following examples demonstrate how to enable the asynchronous response listener when defining the custom port of 9999:

- Use the following Java command to configure the custom HTTP port for the asynchronous response listener in a thin client environment:
  - `java.exe` **`-Dcom.ibm.websphere.webservices.http.listenerPort=9999`** `com.ibm.websphere.my_program`
- Use the following launchClient command to configure the custom HTTP port for the asynchronous response listener in an application client container:
  - `launchClient.bat MyClient.ear` **`-CCDcom.ibm.websphere.webservices.http.listenerPort=9999`**
- The following is an excerpt from an HTTPTransport policy binding.xml file that includes the asynchronous response listener properties:

```
</wsp:Policy>
   </wsp:ExactlyOne>
 </wsp:All>
    <wshttp:outAsyncResponseProxy>
     <wshttp:connectInfo host="" port=""></wshttp:connectInfo>
                  <wshttp:basicAuth userid="" password=""></wshttp:basicAuth>
    </wshttp:outAsyncResponseProxy>
    <wshttp:properties>
  <wshttp:customProperty name="com.ibm.websphere.webservices.http.listenerPort" value="9999" />
   </wshttp:properties>
 </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

What to do next

Run the JAX-WS client with the specified asynchronous response listener options.

Invoking JAX-WS web services asynchronously using the SOAP over JMS transport

Using the JAX-WS JMS asynchronous response message listener

Java API for XML-Based Web Services (JAX-WS) includes a Java Message Service (JMS) asynchronous response message listener, which is used to receive responses to asynchronous JAX-WS requests that use the JMS transport. The JMS asynchronous response message listener is used in the application server and application client environments.

Before you begin

JAX-WS provides support for invoking web service operations asynchronously by using either a callback or a polling model. When the JAX-WS client uses the JMS transport to invoke asynchronous operations, the

responses are received by the asynchronous response message listener. To learn how to use the JAX-WS asynchronous client invocation model, read about invoking JAX-WS web services asynchronously.

About this task

The JMS asynchronous response message listener is used within the web services client environment to receive incoming asynchronous responses when the client application is using the JMS Transport. The listener requires a connection factory and a queue to function correctly. Begin by configuring the connection factory and queue, and then specify the JNDI names of the connection factory and queue to the listener by setting Java system properties. The environment in which the client is running determines how the system properties are set.

The JMS asynchronous response message listener is started automatically by the web services client runtime environment when the client invokes the first asynchronous JAX-WS operation using the JMS transport.

The connection factory and the queue configured with the asynchronous response message listener is used for all requests that are invoked within a particular Java process such as for the application server or an application client container. You can share the connection factory among different Java processes. However, you cannot share a queue among Java processes.

Procedure

1. Determine if you want the JAX-WS client to use the JMS transport mechanism.
2. For each Java process that will use JMS as a transport for asynchronous JAX-WS requests, configure the connection factory and queue that are used by the JMS asynchronous response listener for that process. You can share a connection factory among multiple Java processes, but you cannot share a queue among Java processes.
3. For each Java process, set the com.ibm.websphere.webservices.jms.AsyncReplyQueueName and com.ibm.websphere.webservices.jms.AsyncReplyCFName Java system properties to specify the JNDI names of the queue and connection factory that are used by the JMS asynchronous response message listener for that process.

   If the JNDI name of the queue is the default value, jms/DefaultAsyncReplyQueue, then you do not need to set the AsyncReplyQueueName property. Likewise, if the JNDI name of the connection factory is the default value, `jms/DefaultAsyncReplyCF`, then you do not need to set the AsyncReplyCFName property as well.

   If your client runs within the application server environment, then set the properties as application server system properties by using the administrative console or the **wsadmin** command.

   If your client runs within the application client container environment, then you should set the properties by using the `–CCD` option on the **launchClient** command line.

Results

Your JAX-WS web services client is configured to use the JMS asynchronous response message listener to receive asynchronous response messages when using the JMS transport.

Example

Suppose that you have a JAX-WS web services client that runs in the application client container environment and uses the JMS transport to communicate with the server. Suppose also that the client invokes asynchronous JAX-WS operations. You can create a connection factory with the JNDI name, `jms/MyAppCF`, and a queue with the JNDI name, `jms/MyAppAsyncReplyQueue`. When you invoke the client with the **launchClient** command, specify the JNDI names of the queue and connection factory as illustrated in the following command:

```
launchClient MyAppClient.ear \
   -CCDcom.ibm.websphere.webservices.jms.AsyncReplyQueueName=jms/MyAppReplyQueue \
   -CCDcom.ibm.websphere.webservices.jms.AsyncReplyCFName=jms/MyAppCF \
   <application arguments>
```

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

APACHE INFORMATION. This information may include all or portions of information which IBM obtained under the terms and conditions of the Apache License Version 2.0, January 2004. The information may also consist of voluntary contributions made by many individuals to the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org. You may obtain a copy of the Apache License at http://www.apache.org/licenses/LICENSE-2.0.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Intellectual Property & Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X